
# Test-Case Generation with Automata-based Software Model Checking

Max Barth and Marie-Christine Jakobs

LMU Munich, Munich, Germany
Max.Barth@lmu.de, M.Jakobs@lmu.de

**Abstract.** Software quality is often evaluated by testing the software on an adequate test suite, e.g., a test suite achieving certain or high coverage of the software. Manually generating such test suites is tedious. Thus, several automatic test-case generation approaches were developed to support this task. Approaches based on software model checking typically achieve high coverage and have been shown to be sufficiently efficient in the past. Yet, there does not exist a test-case generation approach that builds upon the automata-based approach to software model checking e.g., successfully used by Ultimate Automizer. To close this methodical gap, we present Ultimate TestGen, a test-case generator built on Ultimate Automizer. An experimental comparison of Ultimate TestGen against a closely related, up-to-date test-case generation approach reveals that Ultimate TestGen generates test suites that achieve the same or higher branch coverage for nearly 75% of the evaluated programs.

## 1 Introduction

Testing is a widely adopted technique to inspect software quality and structural coverage criteria are common metrics to judge the adequacy of generated test suites. Since manually generating test cases or even entire test suites is laborious, several automatic test-case generation techniques have been suggested in the past. Many of them generate test suites for structural coverage criteria, in particular branch coverage, thereby focusing on test input generation. They range from random testing and fuzzing [37,32,33,42] over search-based testing [34] to symbolic execution [16,38] and software model checking [7,41,8,28,29,9].

By design, approaches based on software model checking achieve high coverage because they check the reachability of each individual test goal. While one might think that checking the reachability of test goals is too expensive, advances in verification technology [5] allowed for the development of rather efficient test-case generation approaches like e.g., CoVeriTest [9], FuSeBMC [1], or VeriFuzz [35], which use (bounded) software model checking.

Inspired by the success of these approaches and the success of the software model checker Ultimate Automizer [5], we propose the test-case generation technique Ultimate TestGen. Ultimate TestGen is based on the automata-based approach to software model checking [27] that is employed by Ultimate Automizer [25] and that has not been used for test-case generation before.

To turn the software model checker Ultimate Automizer into a test-case generator, we employ an approach widely-used when generating test cases with verifiers. Namely, we generate test cases from counterexamples [7,41]. Besides transforming counterexamples into tests, we therefore need to encode the hit of a (new) test goal $g$ as a property violation, i.e., Ultimate Automizer must accept program execution traces that reach the test goal $g$. To become efficient, we follow e.g., CoVeriTest [9] and consider multiple test goals at once. Hence, we need Ultimate Automizer to continue verification after detecting a counterexample. Meanwhile, it should avoid reporting counterexamples that only consider already covered test goals. In contrast to existing approaches [21,9,39], which remove test goals, we achieve this by abstracting such counterexample traces by automata and then restricting Ultimate Automizer's exploration to program execution traces that are not accepted by any of those automata.

We implemented the above extensions into Ultimate Automizer. The resulting tool Ultimate TestGen allows us to choose between two configurations: all and incr. Ultimate TestGen-all starts test-case generation with all test goals at once. In contrast, Ultimate TestGen-incr incrementally increases the set of considered test goals, adding more promising test goals earlier. A test goal will be considered more promising if it may cover a higher number of additional test goals when covered. We experimentally compare both configurations on the benchmark programs used in the International Competition on Software Testing (Test-Comp) [6]. Our evaluation shows that Ultimate TestGen-all achieves equal or higher coverage for 97% of the benchmark programs while Ultimate TestGen-incr generates significantly fewer test cases. Furthermore, a comparison with CoVeriTest, an up-to-date Test-Comp participant, which like Ultimate TestGen uses counterexample-guided abstraction refinement, reveals that Ultimate TestGen-all achieves the same coverage for about 60% of the benchmarks and even higher coverage for 5% of the programs. Comparing Ultimate TestGen-all against CoVeriTest's component based on predicate abstraction, the most similar approach to Ultimate TestGen, exhibits that Ultimate TestGen-all even achieves higher coverage for 30% of the benchmark programs.

## 2 The Basics of Software Model Checking with Automata

Our goal is to use an automata-based approach to software model checking [27] to detect feasible error traces (i.e., counterexamples) in programs and then generate tests from them. This section introduces the types of automata used by this approach and fixes the meaning of a feasible error trace.

### 2.1 Program Automata and their Feasible Error Traces

We consider programs that sequentially execute statements from a fixed set $\Sigma$. For our presentation, we assume that $\Sigma$ contains assignments and assume statements over a set $V$ of (integer) variables.[1]. Following literature [27], we then

[1] In our implementation, we support C programs.

```
0: r = 1;
1: while x > 0
2:     r = r*x;
3:     x = x - 1;
4: if x < 0
5:     r = 0;
```

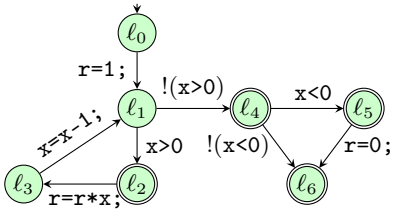


**Fig. 1.** Source code (left) and program automaton (right) of our example program `fac`

model a program together with its correctness property by a (program) automaton $A_{\mathcal{P}} = (L, \delta_{\mathcal{P}}, \ell_0, F_{\mathcal{P}})$. Thereby, the set $L$ represents the program counter values including the initial program counter $\ell_0 \in L$, while $\delta_{\mathcal{P}} \subseteq L \times \Sigma \times L$ denotes the control-flow edges, which describe which statements can be executed at a given program location and where to proceed after the statement's execution. In addition, the set $F_{\mathcal{P}} \subseteq L$ describes the error locations that model the correctness property of a program. More concretely, program executions reaching any of the error locations in $F_{\mathcal{P}}$ violate the program's correctness property. Thus, $F_{\mathcal{P}}$ depends on the property to be checked. Since we aim to generate tests from property violations, we use $F_{\mathcal{P}}$ to characterize the coverage of test goals, in our case program branches (i.e., assume statements in the program automaton).

Figure 1 shows the code and the program automaton of our example program `fac`, which computes the factorial of `x` if `x` is non-negative. The automaton contains one edge per assignment and two assume-statement edges per condition of each if or while statement, namely one for each outcome. Moreover, its error locations are the end locations of the program's assume statements (branches).

To generate tests from property violations, our approach inspects error traces, i.e., sequences of statements that are allowed by the program syntax and lead to a property violation. Formally, a sequence of statements $\pi \in \Sigma^*$ is an *error trace* of a program if its program automaton $A_{\mathcal{P}}$ accepts $\pi$. Since error traces only consider the program's syntax, not all error traces can be observed during program execution. An example of a non-observable error trace is the sequence of statements `r=1;,x>0,r=r*x;,x=x-1;,!(x>0),x<0,r=0;` of program `fac`. Note that it is unobservable because it visits the loop once (`x>0,r=r*x;,x=x-1;,!(x>0)`) and after loop execution, statement `x<0` conflicts with the possible values of variable `x`, which are not smaller than zero. Due to such potential conflicts between variable values and statements, our approach needs to check that an error trace is feasible. An error trace is feasible if there exists a program execution that executes the exact same sequence of statements as the error trace. Since an error trace is syntactically allowed by the program, to show its feasibility, it remains to be proven that it does not conflict with the variable values during execution, i.e., it respects the statements' semantics. A statement's semantics basically defines for which variable values the statement is executable and what the variable values resulting from executing the statement are. We represent the variable values using data states that assign to every program variable a value

of its domain. On top of the set $D$ of data states, we then define the semantics of a statement `stmt` by a (partial) function $SP_{\texttt{stmt}} : D \rightharpoonup D$. For assume statements $\texttt{stmt}_?$, function $SP_{\texttt{stmt}_?}$ is a partial identity function that is defined for data state $d$ if the assume (a Boolean expression) evaluates to true in data state $d$, i.e., $d \models \texttt{stmt}_?$. For assignments $\texttt{stmt}_=$, function $SP_{\texttt{stmt}_=}$ is total and denotes the strongest-post operator of the semantics. Finally, an (error) *trace* $\pi = \texttt{stmt}_1, \ldots, \texttt{stmt}_n \in \Sigma^*$ *is feasible* if there exists a sequence of data states $d_0, d_1, \ldots, d_n$ such that $SP_{\texttt{stmt}_i}(d_{i-1}) = d_i$ holds for all $1 \leq i \leq n$. Technically, we check the feasibility of an error trace $\pi$ by first converting it into single static assignment (SSA) form [20]. Then, we encode its SSA form into a logic formula $\varphi_\pi$ that will be satisfiable if $\pi$ is feasible. Finally, we use a satisfiable modulo theory (SMT) solver to determine the feasibility of $\varphi_\pi$.

### 2.2 Abstracting Traces via Interpolant and Error Automata

To generate a test suite that achieves a high coverage, we intend the software model checker to detect one feasible error trace per test goal, i.e., per error location from $F_{\mathcal{P}}$. To efficiently detect those feasible error traces, we successively exclude irrelevant error traces from the search space of the model checker. More concretely, we exclude error traces for which we know that they (a) are infeasible or (b) do not contribute to the coverage because they end in an already covered test goal. Technically, we build automata $\mathcal{A}_i$ that each accept a subset of irrelevant error traces and subtract them from the program automaton.

Whenever we detect that an error trace $\pi$ is infeasible, we build an *interpolant automaton* [26] that accepts the analyzed, infeasible trace plus some infeasible traces that have a similar reason for infeasibility. Thereby, each interpolant automaton encodes the reason for infeasibility of the accepted traces. Formally, an interpolant automaton $\mathcal{A}_\mathcal{I} = (Q_\mathcal{I}, \delta_\mathcal{I}, q_0, F_\mathcal{I}, d_\mathcal{I})$ consists of a set of states $Q_\mathcal{I}$ including the initial state $q_0$ and the accepting states $F_\mathcal{I}$ as well as a transition relation $\delta_\mathcal{I} \subseteq Q_\mathcal{I} \times \Sigma \times Q_\mathcal{I}$. To record the infeasibility argument of accepted traces, it also contains a total function $d_\mathcal{I} : Q_\mathcal{I} \to 2^D$ that assigns to each state a set of data states that overapproximate the reachable data states while still allowing to prove infeasiblility of accepted traces.[2] To guarantee the infeasibility of accepted traces, each interpolant automaton ensures that (1) the initial state allows any data state $d_\mathcal{I}(q_0) = D$, (2) the transition relation respects the statement's semantics, i.e., $\forall (q, \texttt{stmt}, q') \in \delta : \{d' \mid \exists d \in d_\mathcal{I}(q) : SP_{\texttt{stmt}}(d) = d'\} \subseteq d_\mathcal{I}(q')$, and (3) final states do not allow any data state, i.e., $\forall q_f \in F_\mathcal{I} : d_\mathcal{I}(q_f) = \emptyset$. Together properties (1)–(3) ensure that any accepted trace is infeasible.

Figure 2 shows one interpolant automaton for the infeasible error trace $\pi =$`r=1;,x>0,r=r*x;,x=x-1;,!(x>0),x<0,r=0;` from above, which might be constructed by Ultimate Automizer. The figure uses assertions known from Hoare logic to describe the set of data states assigned to each state. The automaton encodes the error trace into a corresponding sequence of automaton

[2] In practice, the sets of data states are represented by assertions.

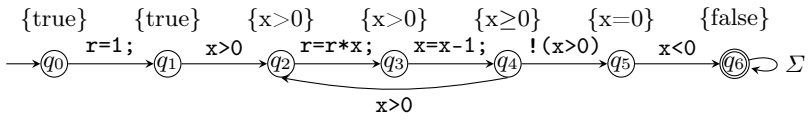


**Fig. 2.** Example of an interpolant automaton

transitions. In addition, the states are annotated with interpolants (the assertions), which are derived by splitting the encoding $\varphi_\pi$ at the position of the state and computing the interpolant. Thereafter, the automaton is enriched by appropriate backward edges resulting from loops that keep the reason of infeasibility. In our case, the automaton accepts any error trace that executes the loop of example program `fac` at least once and thereafter enters the if branch. We refer to [26] for more details on how to construct an interpolant automaton based on the proof of unsatisfiability of an error trace and interpolation.

Whenever we detect a feasible error trace $\pi$, we first generate a test as explained in the next section. The generated test visits all test goals on the error trace. Thus, error traces ending in an error location associated with a visited test goal are irrelevant in future because they only contribute to the coverage when accidentally visiting other uncovered test goals. To exclude such error traces from future analysis, for each of the error locations $\ell_e$ visited by $\pi$ we will construct an *error automaton* if no error automaton already exists for $\ell_e$. Given an error location, the error automaton accepts any trace that ends in that error location. For the construction of the error automaton, we make use of the following structural properties of the error locations that we consider for test-case generation. First, the error locations will have exactly one incoming transition[3]. Second, we will label statements such that the labeled statement on the incoming transition is unique in the program automaton, i.e., the statement does not occur with the same label on any other transition of the program automaton. Hence, we can use the (labeled) statement to detect whether the error location is reached. Based on these assumption, the error automaton for an error location $\ell_e$ with corresponding incoming transition $(\ell, \texttt{stmt}, \ell_e)$ consists of two states, the initial state $q_0$ and the final state $q_e$. If the automaton observes the statement `stmt`, it transitions to the final state $q_e$ and otherwise, it transitions to $q_0$. Formally, the automaton is defined as follows.

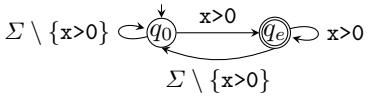


**Fig. 3.** Example of an error automaton

$$\mathcal{A}_E = \left( \{q_0, q_e\}, \begin{array}{l} \{(q_0, \texttt{stmt}, q_e), (q_e, \texttt{stmt}, q_e)\} \cup \\ \{(q, \texttt{stmt}', q_0) \mid (q = q_0 \vee q = q_e) \wedge \texttt{stmt}' \in \Sigma \setminus \{\texttt{stmt}\}\} \end{array}, \{q_0\}, \{q_e\} \right)$$

[3] Note that this property is violated for error location $\ell_6$ of our example program. However, our encoding of test goals, which we explain later, ensures this property.

**Algorithm 1** Ultimate TestGen-all employing automata-based software model checking [26] for test-case generation

**Input:** Program $A_{\mathcal{P}} = (L, \delta_{\mathcal{P}}, \ell_0, \emptyset)$, test goals $G \subseteq \delta_{\mathcal{P}}$
1: $A_{\mathcal{P}_{\text{test}}}$ := encode_test_goals($A_{\mathcal{P}}$, G);
2: `test_suite` := $\emptyset$; $A_{\mathcal{P}_{\text{rel}}} := A_{\mathcal{P}_{\text{test}}}$
3: **while** $\mathcal{L}(A_{\mathcal{P}_{\text{rel}}}) \neq \emptyset$ **do**
4: determine $\pi \in \mathcal{L}(A_{\mathcal{P}_{\text{rel}}})$
5: (result, witness) := check_feasibility($\pi$);
6: **if** result = true **then**
7: test_suite := test_suite $\cup$ generate_test_case($\pi$, witness);
8: $A_\pi$ := generate_error_automata($\pi$);
9: **else**
10: $A_\pi$ := generate_interpolant_automata($\pi$, witness);
11: $A_{\mathcal{P}_{\text{rel}}} := A_{\mathcal{P}_{\text{rel}}} \setminus A_\pi$;
12: return `test_suite`;

Figure 3 shows the error automaton for the feasible error trace `r=1;,x>0` of program `fac`. The next section explains how to use the automata introduced in this section for test-case generation.

## 3 From Model Checking to Test-Case Generation

Our goal is to turn the software model checker Ultimate Automizer into a test-case generator that detects one feasible error trace per test goal and transforms it into a test case. To this end, we adapt its model checking approach.

Before we explain the adaption, let us recapture the original approach [26]. Given a program automaton, Ultimate Automizer iteratively refines an overapproximation of the feasible error traces until the overapproximation is empty, i.e., it proves that the program is correct, or Ultimate Automizer detects a feasible error trace, i.e., it finds a property violation. First, the initial overapproximation becomes the program automaton, which accepts all error traces of the program. Next, each iteration selects an error trace $\pi$ from the current overapproximation and checks its feasibility. If $\pi$ is feasible, a violation is found and $\pi$ is returned as a counterexample. If $\pi$ is infeasible, an interpolant automaton is constructed with the help of the infeasibility proof and the overapproximation is refined. More concretely, the traces accepted by the interpolant automaton are excluded from the overapproximation and then the subsequent iteration starts.

Next, let us discuss how to adapt the above procedure for test-case generation. Algorithm Ultimate TestGen-all (Alg. 1) describes the adapted procedure. For now, let us assume that we already encoded the test goals into the program and let us focus on lines 2–12. The algorithm maintains two important data structures: `test_suite`, which contains the generated test cases, and the overapproximation $A_{\mathcal{P}_{\text{rel}}}$ of feasible and relevant error traces. Like the original approach, the initial overapproximation becomes the program automaton. In

addition, the initial test suite is empty, i.e., no test cases have been generated. Furthermore, the while loop realizes the iterative refinement. At the beginning of each iteration, the algorithm checks whether the overapproximation $A_{\mathcal{P}_{\text{rel}}}$ contains any error trace. Since the overapproximation is described by an automaton, the algorithm checks whether the language of the automaton is empty. If the language is empty, no relevant error traces exist, i.e., all error locations are covered[4], and we return the test suite. Otherwise, we perform the next loop iteration. First, we use an $A^*$ algorithm [23,24] to determine a relevant error trace $\pi$ of the program (line 4) and, then, check $\pi$'s feasibility as explained in the previous section. The feasibility check returns the result and a witness, a model in case of feasibility and the unsatisfiability proof otherwise. If error trace $\pi$ is feasible, we deviate from the original approach. Instead of returning $\pi$ and reporting a property violation, we generate a test case from $\pi$ and the witness (a model). Afterwards, we generate the error automaton $A_\pi$ for the error trace $\pi$ as described in the previous section. Remember the error automaton accepts all traces that end in the same error location as $\pi$ and, thus, helps us to avoid that traces that end in already covered error locations (test goals) are found in future. If $\pi$ is infeasible, we proceed as the original approach and generate an interpolant automaton $A_\pi$ from $\pi$ and the witness (a proof of unsatisfiability). In both cases, we refine the overapproximation by subtracting the generated automaton $A_\pi$ and continue with the next iteration.

**Generating Test Cases for Feasible Error Traces** After describing the test-case generation procedure, we now explain in detail how to derive a test case from a feasible error trace. Like many other tools [7,16,9,1,35], our test cases only provide test inputs. Hence, they must specify the values for input parameters and external functions like e.g., `random`, `scanf`. To compute these input values, we use an approach similar to the one of Blast [7]. For the feasibility check, we already computed a SSA-based formula encoding $\varphi_\pi$ of the error trace $\pi$. Since we generate test cases from feasible error traces, the witness returned by the feasibility check is a model of $\varphi_\pi$. To generate the test case, we only need to identify the variables in the formula $\varphi_\pi$ that correspond to inputs, look up their values in the model, and export their values in the order of the variables' occurrence in $\varphi_\pi$. For the export, we utilize the format[5] used by the International Competition on Software Testing [6], which allows test execution with TestCov [13]. For example, consider error trace $\pi$ =`r=1;,x>0` and corresponding formula $\varphi_\pi = r_1 = 1 \wedge x_0 > 0$. The first and only input in $\varphi_\pi$ is $x_0$. Given the model $\{r_1 \mapsto 1, x_0 \mapsto 2\}$ of $\varphi_\pi$, we derive the following test case:

```
<testcase><input>2</input></testcase>
```

[4] We encode the test goals into the program such that all test goals will be covered if all error locations are covered. Hence, we will abort if all test goals are covered.

[5] https://gitlab.com/sosy-lab/test-comp/test-format/-/blob/main/doc/Format.md

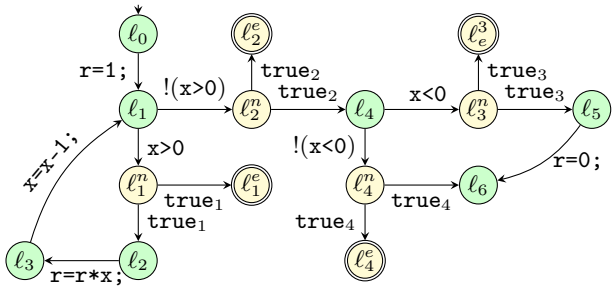


**Fig. 4.** Program automaton resulting from encoding branch coverage into our example

**Encoding Test Goals** So far, we assumed that the test goals are already encoded into the program. Next, we explain how to actually encode them, i.e., we explain line 1 of Alg. 1. Like e.g., CoVeriTest [9], we are interested in structural coverage, in particular branch coverage, that can be expressed as the coverage of a set $G \subseteq \delta_{\mathcal{P}}$ of control-flow edges. For branch coverage, the set $G$ of test goals contains all control-flow edges with an assume statement. However, control-flow edges are not the appropriate format to specify property violations because program automata specify property violations with error locations. Thus, we need to perform a testability transformation [22] and translate the coverage of control-flow edges into the coverage of error locations.

A naive translation would use the predecessors or successors of the test goal edges as error locations. For branch coverage, the predecessors are decision nodes forking into two assume statements (i.e., test goals) and are, thus, inappropriate. Next, we show that successors are inappropriate as well. For this, consider Fig. 1. We observe that the coverage of assume statement `!(x<0)` is associated with error location $\ell_6$. However, location $\ell_6$ can be reached via the incoming edge with statement `r=0;` without executing the assume statement `!(x<0)`. Hence, we require a more intricate transformation.

To ensure that test goal coverage is equivalent to coverage of error locations, our idea is to replace any edge $g = (\ell, \texttt{stmt}, \ell')$ in the test goals by two successive edges $(\ell, \texttt{stmt}, \ell_g^n), (\ell_g^n, \texttt{true}, \ell')$, thereby introducing a new (intermediate) location $\ell_g^n$. Since an assume statement does not change the data state and `assume true` is satisfied by any data state, the two successive edges $(\ell, \texttt{stmt}, \ell_g^n), (\ell_g^n, \texttt{true}, \ell')$ are semantically identical to $g$. While we could use locations $\ell_g^n$ as error locations, constructing appropriate error automata is rather complex. The error automata from the previous section require unique statements. Thus, we further extend our translation. First, we use labeled assumptions $\texttt{true}_g$, which are semantically equivalent to assumption `true`, but make them unique statements and allow us to use the error automata from the previous section. While this is sufficient, it forces an exploration algorithm to either detect shallow goals first and, thus, miss the potential of covering multiple goals at once, or be able to leave accepting states during exploration. To overcome

this, we use non-determinism. More concretely, we introduce an additional edge $(\ell_g^n, \texttt{true}_g, \ell_g^e)$ and use $\ell_g^e$ instead of $\ell_g^n$ as error location. This leads us to the following transformation.

**Definition 1.** *Given program automaton* $A_{\mathcal{P}} = (L, \delta_{\mathcal{P}}, \ell_0, \emptyset)$ *and test goals* $G \subseteq \delta_{\mathcal{P}}$*, we define* `encode_test_goals`$(A_{\mathcal{P}}, G) = (L_{\mathcal{P}_{test}}, \delta_{\mathcal{P}_{test}}, \ell_0, F_{\mathcal{P}_{test}})$*, where*

- $F_{\mathcal{P}_{test}} = \{\ell_g^e \mid g \in G\}$,
- $L_{\mathcal{P}_{test}} = L \cup F_{\mathcal{P}_{test}} \cup \{\ell_g^n \mid g \in G\}$*, and*
- $\delta_{\mathcal{P}_{test}} = (\delta_{\mathcal{P}} \setminus G) \cup \{(\ell_g^n, \texttt{true}_g, \ell_g^e), (\ell, \texttt{stmt}, \ell_g^n), (\ell_g^n, \texttt{true}_g, \ell') \mid (\ell, \texttt{stmt}, \ell') \in G\}$.

Figure 4 shows the transformed program automaton $A_{\mathcal{P}_{\text{test}}}$ for our example program `fac` from Fig. 1. The newly added states are highlighted in yellow. Note that like our implementation, Fig. 4 uses unique integer numbers to name the locations instead of the edges $g$ used in the above formalization.

**Towards Exploiting Collateral Coverage** Often, shorter error traces are prefixes of longer error traces. For example, `r=1;,!(x>0),true`$_2$ is a prefix of `r=1;,!(x>0),true`$_2$`,x<0,true`$_3$. In addition, a test case that covers the longer trace implicitly covers its shorter prefixes and, thus, may cover additional test goals like `!(x>0)`, which is known as collateral coverage. Hence, covering deeper test goals (error locations) first and considering collateral coverage may reduce the size of the test suite and potentially also the test-case generation time. However, proving unreachability of an error location can be impossible and even if possible it is typically more expensive than detecting a feasible error trace. Therefore, we should not indefinitely focus on a certain subset of test goals. Our suggestion to the problem is to incrementally increase the set of considered goals, thereby adding deeper goals first.

Algorithm Ultimate TestGen-incr shown in Alg. 2 describes test-case generation with an incrementally increasing goal set. To this end, Ultimate TestGen-incr uses an additional data structure `goals` to keep track of the goals that need to be added. Initially, all goals need to be added. Furthermore, Ultimate TestGen-incr extends the test-case generation loop. At the beginning of each iteration, it adds a new goal in case unconsidered goals still exist. Our implementation aims to select deeper goals (error locations) first. Since deeper states have higher IDs, we select the goal (error location) with the highest ID. To ensure that we only consider traces that are accepted by one of the currently considered goal states, we extend the error trace selection with a constraint on the selected trace. Due to the design of the test goal encoding, our constraint achieves the desired purpose. Note that we could have used a similar constraint to exclude traces to already covered error locations. However, we believe that our automata-based exclusion is better because it reduces the state space. Getting back to the discussion of our additions, we also consider the collateral coverage when generating a test case from a feasible error trace. Note that by design of the test goal encoding, we will cover test goal $g$ if we execute statement $\texttt{true}_g$ and every prefix of the error trace $\pi$ that ends in a statement $\texttt{true}_g$ is an error trace with corresponding error location $\ell_g^e$. We use this insight to compute multiple error automata that also exclude covering collaterally covered test goals. To this

**Algorithm 2** ULTIMATE TESTGEN-INCR extending test-case generation with support for collateral coverage

**Input:** Program $A_{\mathcal{P}} = (L, \delta_{\mathcal{P}}, \ell_0, \emptyset)$, test goals $G \subseteq \delta_{\mathcal{P}}$

```
 1: A_{P_test} := encode_test_goals(A_P, G);
 2: test_suite := ∅; A_{P_rel} := A_{P_test}; goals := F_{P_test};
 3: while goals≠ ∅ ∨ L(A_{P_rel}) ≠ ∅ do
 4:    if goals≠ ∅ then
 5:       pop ℓ^e_g from goals;
 6:    select π ∈ A_{P_rel} ending with true_g s.t. ℓ^e_g ∈ F_{P_test} \ goals;
 7:    (result, witness) := check_feasibility(π);
 8:    if result = true then
 9:       test_suite := test_suite ∪ generate_test_case(π, witness);
10:       A_π := ⋃_{π_p ∈ error_prefixes(π)} generate_error_automata(π_p);
11:       goals := goals\{ℓ^e_g | true_g occurs in π};
12:    else
13:       A_π := generate_interpolant_automata(π, witness);
14:    A_{P_rel} := A_{P_rel} \ A_π;
15: return test_suite;
```

end, line 10 computes the union of all error automata built for all prefixes $\pi_p$ of the error trace $\pi$ that are error traces themselves.[6] The last addition in ULTIMATE TESTGEN-INCR is line 11, which excludes all error locations corresponding to collaterally covered test goals from the set of unconsidered test goals.

**Implementation** We realized Alg. 1 and 2 in the verifier ULTIMATE AUTOMIZER [25], which already supports automata-based software model checking. ULTIMATE AUTOMIZER uses an $A^*$ algorithm [23,24] to detect error traces. We do not provide $A^*$ with a specific heuristic, but ULTIMATE TESTGEN-INCR guides it with the error locations to consider (i.e., $F_{\mathcal{P}_{\text{test}}} \setminus \texttt{goals}$). Also, ULTIMATE AUTOMIZER already provides the automata operations, the construction of interpolant and error automata, and the feasibility check for error traces.

To encode the test goals, we extend ULTIMATE AUTOMIZER's translation frontend, which translates C programs into Boogie programs and then builds the program automaton. While our approach allows arbitrary test goals, our implementation only supports branch coverage, i.e., the test goals are all edges with assume statements. To encode these test goals, we add an `assert false` statement with a unique label at the beginning of each branch in the Boogie program. When the program automaton is built, every assert statement is translated into two edges. One edge represents the violation of the assert statement (i.e., `!false`≡`true` is true), ends in an error location and corresponds to $(\ell_g^n, \texttt{true}_g, \ell_g^e)$. The other edge passes the assertion and corresponds to $(\ell_g^n, \texttt{true}_g, \ell')$. Since `assert false` cannot be passed, we configure ULTIMATE AUTOMIZER to ignore the assert con-

[6] For efficiency, our implementation computes one single automata for all error traces that is equivalent to the union of their error automata.

dition when passing an assert statement, which is equivalent to condition `true`. Furthermore, Ultimate Automizer uses block encoding, which assigns loop free code blocks instead of statements to edges. To ensure that we do not loose inputs due to this encoding, we change Ultimate TestGen's block encoding and interrupt a block if an input occurs in the program.

For the feasibility check, Ultimate Automizer first uses unbounded integers, as known from mathematics, and a combination of the SMT solvers Z3 [36] and SMTInterpol [18]. If the feasibility check returns true and the encoding is not precise because the encoded trace considers floats, doubles, bit-wise operations, etc., we will stop test-case generation with unbounded, mathematical integers and start test-case generation using a bit-vector encoding, which uses the two SMT solvers MathSAT5 [19] and CVC4 [3]. To generate a test case for a feasible error trace, we use the model provided by the feasibility check and proceed as described above. However, due to the use of e.g., unbounded integers, the values might be out of range of the variable's C type. Therefore, we identify the required C type first and if a value is out of range, we use the value modulo the allowed size. If there exists a variable whose value is out of range and the size of the C type depends on the architecture, we write two test cases. One test case shrinks the values to sizes appropriate for 32-bit architectures and the other for 64-bit architectures.

## 4 Evaluation

In our evaluation, we aim to investigate the following two research questions.

**RQ1** How do the two configurations of Ultimate TestGen compare in terms of achieved coverage and number of generated test cases?

**RQ2** How does Ultimate TestGen compare to similar, up-to-date competitors in terms of achieved coverage and number of generated test cases?

### 4.1 Evaluation Setup

**Tasks** For our evaluation, we consider the coverage criterion branch coverage and perform the evaluation on the corresponding 2 933 tasks considered in the International Competition on Software Testing (Test-Comp) in 2023 [6].

**Tools** We consider the two test-case generation techniques presented in this paper, which are implemented in Ultimate Automizer (TestGeneration branch[7] version 94ac8e0). As competitors, we considered the closely related test-case generators from Test-Comp 2023 [6], i.e., test-case generators that also perform counterexample-guided abstraction refinement (CEGAR) and predicate abstraction. From those, we selected the one that achieved the highest score in the category `cover-branches`, namely CoVeriTest [9]. CoVeriTest is based on the

[7] https://github.com/ultimate-pa/ultimate/tree/TestGeneration

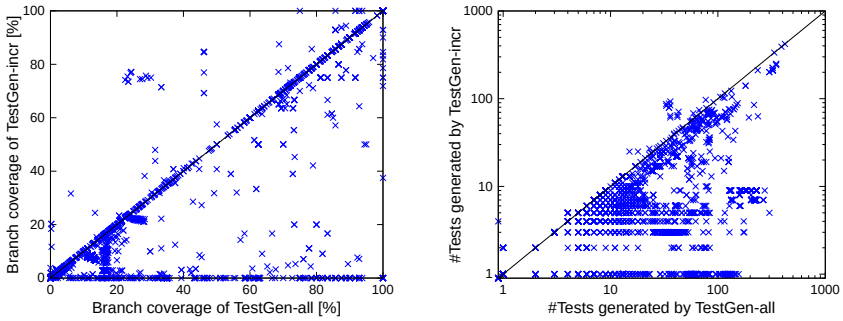


**Fig. 5.** Comparing achieved branch coverage (left) and number of generated test cases (right) of Ultimate TestGen-all (x-axis) and Ultimate TestGen-incr (y-axis)

software analysis framework CPAchecker [10] and uses a portfolio approach that mainly runs a cyclic combination of predicate abstraction [11] and explicit model checking [14]. In addition, it mutates the test cases generated by the cyclic combination and briefly generates random test cases at the beginning. To compare Ultimate TestGen against an even more closely related approach, we also consider CoVeriTest's predicate analysis component (named Predicate) standalone. For both competitors, we use the CoVeriTest version submitted to Test-Comp 2023. Furthermore, we measure the coverage of the generated test suites with TestCov [13] and use the same version as Test-Comp 2023 [6].

**Environment** We run all experiments on machines with 33 GB of memory, an Intel Xeon E3-1230 v5 CPU with 8 processing units and a frequency of 3.40 GHz that run Ubuntu 22.04 (Linux kernel 5.15.0). Following Test-Comp, we use BenchExec 3.18 [15] to limit each test-case generation run to 8 cores, 15 min of CPU time and 15 GB of memory and the TestCov runs to 2 cores, 10 min of CPU time, and 7 GB of memory.

### 4.2 RQ 1: Comparison of Ultimate TestGen Configurations

We aim to compare the two configurations of Ultimate TestGen based on their achieved coverage and the number of generated test cases. In general, we aim for high coverage with a small number of test cases. Figure 5 shows two scatter plots. Its left scatter plot compares for each test task the branch coverage achieved by Ultimate TestGen-all (x-axis) with the branch coverage achieved by Ultimate TestGen-incr (y-axis). We observe that a large number of data points is in the lower right half, i.e., Ultimate TestGen-all achieves a higher coverage. A detailed analysis reveals that Ultimate TestGen-all achieves a higher coverage for 1 055 of 2 933 (36%) tasks and the same coverage for 1 778 of 2 933 (61%) tasks. The reason is that Ultimate TestGen-incr detects longer error traces at the beginning and if they are infeasible, their interpolant automata are larger, too. Large interpolant automata may prohibit efficient program abstractions and

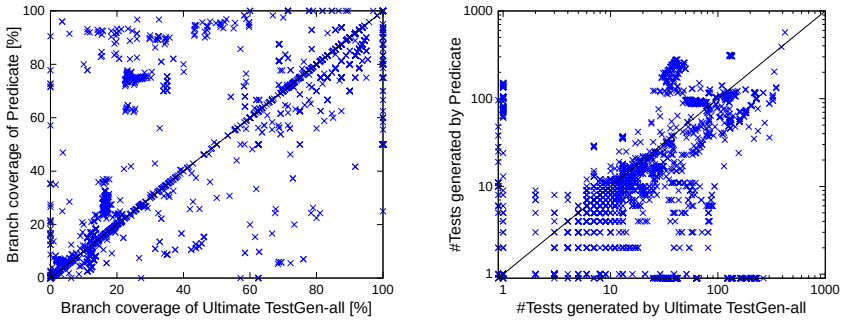


**Fig. 6.** Comparing achieved branch coverage (left) and number of generated test cases (right) of Ultimate TestGen-all (x-axis) and Predicate (y-axis)

automata operations are more expensive on larger automata. Thus, Ultimate TestGen-incr likely detects less feasible error traces in total.

Next, we study the number of generated test cases. The right scatter plot of Fig. 5 compares for each test task the number of test cases generated by Ultimate TestGen-all (x-axis) with the number of test cases generated by Ultimate TestGen-incr (y-axis). We observe that a large number of data points is in the lower right half, i.e., Ultimate TestGen-incr generates fewer test cases. A detailed analysis reveals that Ultimate TestGen-incr generates fewer test cases for 2 258 of 2 933 (77%) tasks and the same number of test cases for 638 of 2 933 (22%) tasks. More importantly for 41% (1 206 of 2 933) of all test tasks, Ultimate TestGen-incr covers at least as many branches as Ultimate TestGen-all, but it generates smaller test suites for 64% (1 206 of 1 878) for them.

In sum, Ultimate TestGen-all achieves better coverage at the cost of more tests but about half of the time Ultimate TestGen-incr is a valuable alternative.

### 4.3 RQ 2: Comparison with Competitors

Next, we compare Ultimate TestGen-all, the better of our two approaches, with the closely related up-to-date competitors Predicate and CoVeriTest. Again, we consider their achieved coverage and the number of generated tests. First, let us compare against Predicate, which is the approach most closely related to ours. The left scatter plot of Fig. 6 compares for each test task the branch coverage achieved by Ultimate TestGen-all (x-axis) with the branch coverage achieved by Predicate (y-axis). We observe that many data points are in the upper left as well as the lower right half, i.e., there exist many tasks for which Ultimate TestGen-all achieves higher coverage and vice versa. Indeed, both approaches achieve the same coverage for 44% (1 298 of 2 933) of the tasks. Predicate achieves a higher coverage for 30% (867 of 2 933) of the tasks, while Ultimate TestGen-all achieves higher coverage for 26% (768 of 2 933) of the tasks. To better understand their strengths and weaknesses, let us study Tab. 1. For each category considered in Test-Comp, Tab. 1 contains a row that shows the

**Table 1.** Sum of achieved coverage per task category and test-case generator

| Category | # Tasks | Ultimate TestGen all | Ultimate TestGen incr | Predicate | CoVeriTest |
|---|---|---|---|---|---|
| Arrays | 292 | 20600 | 18100 | 19500 | 20800 |
| BitVectors | 61 | 3750 | 3350 | 4760 | 4820 |
| ControlFlow | 11 | 32.4 | 27 | 26.6 | 46.1 |
| ECA | 29 | 296 | 267 | 488 | 562 |
| Floats | 197 | 8800 | 8150 | 8820 | 9230 |
| Heap | 110 | 7360 | 4940 | 6850 | 7580 |
| Loops | 661 | 52000 | 47600 | 49300 | 52600 |
| ProductLines | 263 | 5120 | 3010 | 7050 | 7670 |
| Recursive | 51 | 4100 | 3620 | 3730 | 4000 |
| Sequentialized | 91 | 2790 | 45.8 | 7380 | 7350 |
| XCSP | 114 | 11100 | 10700 | 6380 | 11400 |
| Combinations | 671 | 17000 | 14800 | 22500 | 23800 |
| BusyBox | 62 | 982 | 690 | 607 | 1030 |
| DriversLinux64 | 287 | 5830 | 5790 | 5840 | 5920 |
| SQLite | 1 | – | – | – | 0.02 |
| Termination | 32 | 2990 | 2910 | 2800 | 3060 |
| Total | 2933 | 142750.4 | 123999.8 | 146031.6 | 159868.12 |

number of tasks in the category and more importantly for each considered test-case generator the accumulated coverage, i.e., the sum of the coverage achieved for each task in that category. The last row shows the sum of the previous rows. For each row, Tab. 1 highlights the entry with the highest accumulated coverage in light gray. Studying Tab. 1, we notice that there exist some categories like e.g., `BitVectors`, `ECA`, `ProductLines`, `Sequentialized`, or `Combinations`, in which Ultimate TestGen-all achieves a significantly lower accumulated coverage than Predicate. Looking at the SV-COMP results[8], we can recognize that Ultimate Automizer also struggles with the verification tasks of these categories. It seems these categories are particular difficult for Ultimate Automizer. However, there also exist categories like e.g., `Arrays`, `Heap`, `Loops`, `Recursive`, `XCSP`, and `BusyBox` for which Ultimate TestGen-all performs significantly better than Predicate. This is one reason why CPAchecker and CoVeriTest usually use a combination of analyses.

Next, let us compare their numbers of generated test cases. The right scatter plot of Fig. 6 compares this. Again, we observe that many data points are in the upper left as well as the lower right half, i.e., there exist many tasks for which Ultimate TestGen-all generates fewer test cases and vice versa. Still, Predicate generates fewer test cases more often, namely for 68% (2 004 of 2 933 of the tasks). In addition, in 58% (1 263 of 2 165) of the tasks for which Predicate achieves the same or more coverage, Predicate also generates smaller test suites. Nevertheless, Ultimate TestGen-all is complementary to Predicate.

[8] https://sv-comp.sosy-lab.org/2023/results/results-verified/

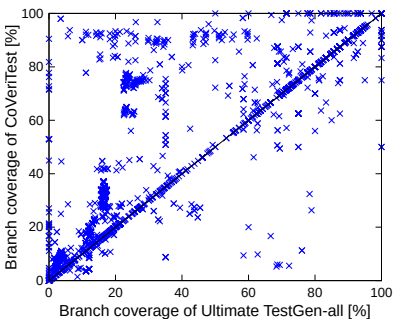


**Fig. 7.** Comparing achieved branch coverage of Ultimate TestGen-all (x-axis) against achieved branch coverage of CoVeriTest

Finally, let us compare Ultimate TestGen-all against CoVeriTest. The scatter plot of Fig. 7 compares their achieved branch coverage. Once more, we observe that there exist data points in the upper left as well as the lower right half, i.e., there exist tasks for which Ultimate TestGen-all achieves higher coverage and vice versa. While CoVeriTest achieves the same coverage in 59% of the cases (1 725 of 2 933 tasks) and even better coverage in 36% of the cases (1 044 of 2 933) of the tasks, Ultimate TestGen-all sometimes achieves better coverage, too. Again, looking at Tab. 1, we notice that there exist some categories like e.g., `BitVectors`, `ECA`, `ProductLines`, `Sequentialized`, or `Combinations`, in which Ultimate TestGen-all achieves a significantly lower accumulated coverage as CoVeriTest. These are mainly the same categories for which Predicate already outperforms Ultimate TestGen-all. In many other categories, Ultimate TestGen-all achieves a similar accumulated coverage and in category `Recursive` it even achieves a higher accumulated coverage. When studying the number of generated test cases, our raw data reveals that CoVeriTest generates more test cases for 76% (2 232 of 2 933) of the tasks because it mutates every test case generated by its model checkers several times. To sum up, Ultimate TestGen complements related state-of-the-art approaches.

### 4.4 Threats to Validity

Our results may not generalize for several reasons. First, we use benchmark programs in our evaluation. Although the benchmark is well-established and contains diverse programs, it may not reflect the characteristics of (all) real-world applications. Second, we used fixed resource limits for the execution of the tasks. Different resource limits may change the results. Third, we compared against two closely related competitors. We expect different results when comparing Ultimate TestGen to more diverse test-case generators.

Also, the coverage results might be imprecise. On the one hand, TestCov may contain bugs that result in wrong coverage numbers. However, TestCov will make the same mistakes for all approaches. In addition, TestCov is an established validator of Test-Comp. Therefore, we expect that significant bugs would have been detected already. On the other hand, TestCov sometimes runs out of resources and may not execute all tests. This happens rarely for Ultimate TestGen and sometimes for Predicate and CoVeriTest. We believe the comparison of the two Ultimate TestGen approaches is hardly affected by this. Moreover, Predicate and CoVeriTest still regularly outperform Ultimate TestGen. Hence, these observations are not affected by the exhaustion of

resource limits. Also, the comparison of the number of generated test cases is not affected by any imprecision of the coverage results.

## 5 Related Work

Symbolic execution is a well-known test-case generation technique [16,17,30,38] based on verification. Moreover, several approaches have been proposed that generate test cases from counterexamples produced by software model checkers. Some of the early approaches are PathFinder [41] and Blast [7]. PathFinder applies explicit-state model checking or symbolic execution. In contrast, Blast uses predicate abstraction to generate test cases. Also, CPA/Tiger [8,39], abstraction-driven concolic testing [21], and CoVeriTest [9] use predicate abstraction. Other test-case generation techniques like FShell [28], CBMC [31], FuSeBMC [1], and VeriFuzz [35] employ bounded model checking. Furthermore, the conditional tester $\texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}$ (Fig. 14 in [12]) describes a template construction to turn an arbitrary verifier into a test-case generator, but we are the first that generate test-cases using an automata-based approach to software model checking [26,27].

Next to their difference in the applied software model checking technique, the approaches also differ in their encoding of the test goals. Test goals in Blast [7] are pairs of program location and target predicate. Abstraction-concolic testing [21] and CoVeriTest [9] describe test goals with a set of control-flow edges and monitor the reachability of those edges. FShell [28] and CPA/Tiger [8,39] express test goals in FQL [29], in particular its representation as test goal automata. FShell encodes the automata into the program, while CPA/Tiger runs the automata in parallel to the analysis. FuSeBMC [1,2] represents test goals by labels in the program. The conditional tester $\texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}$ [12] adds calls to function `__VERIFIER_error()` into the program such that the function is called whenever a test goal is reached. Similarly, our approach adds additional (assert) statements to the program and then characterizes test goals by error locations reachable when executing the statement, i.e., violating the assertion.

Like Ultimate TestGen-all (Alg. 1), FShell [28], abstraction concolic testing [21], CoVeriTest [9], and the conditional tester $\texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}$ [12] consider all test goals at once and exclude already covered goals from being found again. To this end, CoVeriTest and conditional tester $\texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}$ remove covered goals from their specification, while FShell adds SAT constraints on the paths to be detected. In contrast, Ultimate TestGen uses error automata. Similarly, symbolic execution tools like e.g., KLEE [16], which aim to cover every program path, may use coverage-optimized search strategies that prefer states that likely lead to the coverage of new code. At the other extreme, Blast [7] and the BMC component of FuSeBMC [2] aim to cover one test goal at a time. Thereby, Blast considers deeper goals first and FuSeBMC prefers deeper goals but may additionally consider the type of the goal, e.g., whether it is a branch of an if statement or a loop. CPA/Tiger [39] considers a compromise between the two extrema and partitions the set of test goals into subsets either randomly or aiming to cluster test goals with similar prefixes. Goals of one subset are considered

at once and covered goals are removed. If CPA/Tiger cyclically runs multiple analyses [40], the uncovered test goals are repartitioned before each analysis run. Ultimate TestGen-incr (Alg. 2) incrementally increases the considered test goals, thereby prioritizing deeper test goals.

## 6 Conclusion

Testing is a well-established process for quality assurance, which can be supported by automatic test-case generation approaches. We propose Ultimate TestGen, the first test-case generator based on the automata-based approach to software model checking used by Ultimate Automizer. Ultimate TestGen first extends the program automaton, the program representation, to encode the reachability of test goals as property violations. Then, it runs the automata-based verification and transforms feasible counterexamples into test cases. While verification typically stops after detecting a feasible counterexample, Ultimate TestGen aims to cover more than one test goal and, thus, continues verification to detect further counterexamples. To avoid detecting a counterexample triggered by an already covered test goal, Ultimate TestGen extends the verification approach with error automata that allow us to exclude counterexamples triggered by an already covered test goal. Moreover, Ultimate TestGen can be configured to either consider all test goals at once or to incrementally add test goals in the decreasing order of their distance to the initial program location. Our experiments reveal that configuration Ultimate TestGen-all regularly achieves higher coverage while configuration Ultimate TestGen-incr generates smaller test suites. Also, we show that in 70% of the evaluated tasks Ultimate TestGen-all achieves equal or higher coverage than the most similar competitor Predicate. However, it is rarely better than competitor CoVeriTest, which combines different approaches.

**Data-Availability Statement** All experimental data, all used software as well as the test tasks are publicly available in our supplementary aritfact [4].

## References

1. Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in C programs. In: Proc. TAP. pp. 85–105. LNCS 12740, Springer (2021). https://doi.org/10.1007/978-3-030-79379-1_6
2. Alshmrany, K.M., Aldughaim, M., Bhayat, A., Shmarov, F., Aljaafari, F., Cordeiro, L.C.: FuSeBMC v4: Improving code coverage with smart seeds via fuzzing and static analysis. CoRR **abs/2206.14068** (2022). https://doi.org/10.48550/arXiv.2206.14068
3. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proc. CAV. pp. 171–177. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14

4. Barth, M., Jakobs, M.: Replication package for paper "Test-case generation with automata-based software model checking" SPIN 24 (2024). https://doi.org/10.5281/zenodo.10574234
5. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS. pp. 495–522. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29
6. Beyer, D.: Software testing: 5th comparative evaluation: Test-Comp 2023. In: Proc. FASE. pp. 309–323. LNCS 13991, Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_17
7. Beyer, D., Chlipala, A., 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
8. Beyer, D., Holzer, A., Tautschnig, M., Veith, H.: Information reuse for multi-goal reachability analyses. In: Proc. ESOP. pp. 472–491. LNCS 7792, Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_26
9. Beyer, D., Jakobs, M.: 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
10. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
11. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. IEEE (2010), https://ieeexplore.ieee.org/document/5770949/
12. Beyer, D., Lemberger, T.: Conditional testing - off-the-shelf combination of test-case generators. In: Proc. ATVA. pp. 189–208. LNCS 11781, Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_11
13. Beyer, D., Lemberger, T.: TestCov: Robust test-suite execution and coverage measurement. In: Proc. ASE. pp. 1074–1077. IEEE (2019). https://doi.org/10.1109/ASE.2019.00105
14. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE. pp. 146–162. LNCS 7793, Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_11
15. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. STTT **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
16. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
17. Chalupa, M., Vitovská, M., Strejcek, J.: SYMBIOTIC 5: Boosted instrumentation - (competition contribution). In: Proc. TACAS. pp. 442–446. LNCS 10806, Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_29
18. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: Proc. SPIN. pp. 248–254. LNCS 7385, Springer (2012). https://doi.org/10.1007/978-3-642-31759-0_19
19. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS. pp. 93–107. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
20. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. TOPLAS **13**(4), 451–490 (1991). https://doi.org/10.1145/115372.115320

21. 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
22. Harman, M., Hu, L., Hierons, R.M., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE TSE **30**(1), 3–16 (2004). https://doi.org/10.1109/TSE.2004.1265732
23. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. TSSC **4**(2), 100–107 (1968). https://doi.org/10.1109/TSSC.1968.300136
24. Hart, P.E., Nilsson, N.J., Raphael, B.: Correction to "a formal basis for the heuristic determination of minimum cost paths". SIGART Newsl. **37**, 28–29 (1972). https://doi.org/10.1145/1056777.1056779
25. Heizmann, M., Barth, M., Dietsch, D., Fichtner, L., Hoenicke, J., Klumpp, D., Naouar, M., Schindler, T., Schüssele, F., Podelski, A.: Ultimate Automizer and the CommuHash normal form - (competition contribution). In: Proc. TACAS. pp. 577–581. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_39
26. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Proc. SAS. pp. 69–85. LNCS 5673, Springer (2009). https://doi.org/10.1007/978-3-642-03237-0_7
27. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
28. 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
29. 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
30. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
31. Kroening, D., Schrammel, P., Tautschnig, M.: CBMC: the C bounded model checker. CoRR **abs/2302.02384** (2023). https://doi.org/10.48550/arXiv.2302.02384
32. Lemberger, T.: Plain random test generation with PRTest. STTT **23**(6), 871–873 (2021). https://doi.org/10.1007/s10009-020-00568-x
33. Li, J., Zhao, B., Zhang, C.: Fuzzing: A survey. Cybersecurity **1**(1), 6 (Jun 2018). https://doi.org/10.1186/s42400-018-0002-y
34. McMinn, P.: Search-based software test data generation: A survey. STVR **14**(2), 105–156 (2004). https://doi.org/10.1002/stvr.294
35. Metta, R., Medicherla, R.K., Karmarkar, H.: VeriFuzz: Good seeds for fuzzing (competition contribution). In: Proc. FASE. pp. 341–346. LNCS 13241, Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_20
36. de Moura, L.M., Bjørner, N.S.: Z3: An efficient SMT solver. In: Proc. TACAS. pp. 337–340. LNCS 4963, Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
37. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: Proc. ICSE. pp. 75–84. IEEE (2007). https://doi.org/10.1109/ICSE.2007.37
38. Pasareanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. STTT **11**(4), 339–353 (2009). https://doi.org/10.1007/s10009-009-0118-1

39. Ruland, S., Lochau, M., Fehse, O., Schürr, A.: CPA/Tiger-MGP: Test-goal set partitioning for efficient multi-goal test-suite generation. STTT **23**(6), 853–856 (2021). https://doi.org/10.1007/s10009-020-00574-z
40. Ruland, S., Lochau, M., Jakobs, M.: HybridTiger: Hybrid model checking and domination-based partitioning for efficient multi-goal test-suite generation (competition contribution). In: Proc. FASE. pp. 520–524. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_26
41. 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
42. Zeller, A., Gopinath, R., Böhme, M., Fraser, G., Holler, C.: The Fuzzing Book. CISPA Helmholtz Center for Information Security (2023), https://www.fuzzingbook.org/, retrieved 2023-01-07 14:37:57+01:00