International Journal on Software Tools for Technology Transfer (2025) 26:633–646
https://doi.org/10.1007/s10009-024-00769-8

COMPETITIONS AND CHALLENGES

Regular

Six years later: testing vs. model checking

Dirk Beyer[1] · Thomas Lemberger[1]

Accepted: 7 November 2024 / Published online: 16 January 2025

Abstract

Six years ago, we performed the first large-scale comparison of automated test generators and software model checkers with respect to bug-finding capabilities on a benchmark set with 5693 C programs. Since then, the International Competition on Software Testing (Test-Comp) has established standardized formats and community-agreed rules for the experimental comparison of test generators. With this new context, it is time to revisit our initial question: Model checkers or test generators—which tools are more effective in finding bugs in software? To answer this, we perform a comparative analysis on the tools and existing data published by two competitions, the International Competition on Software Verification (SV-COMP) and Test-Comp. The results provide two insights: (1) Almost all test generators that participate in Test-Comp use hybrid approaches that include formal methods, and (2) although the considered model checkers are still highly competitive, they are now outperformed by the bug-finding capabilities of the considered test generators.

Keywords Software verification · Model checking · Program analysis · Test generation · Testing · Fuzzing

1 Introduction

In previous research [32], we compared the bug-finding capabilities of automated test generators and software model checkers on C programs. At the time of that work, no standardized formats existed for the experimental comparison of test generators. So we selected formats for the expected inputs and outputs of test generation, implemented matching adapters for existing test generators, and our own coverage measurement. Nowadays, this is unnecessary. The International Competition on Software Testing (Test-Comp) [19] provides a community-set framework for the evaluation of test generators for the C language, including an exchange format for test suites, a large and well-defined benchmark task set, and agreed-upon resource limitations for benchmarking. So far, the benchmark test tasks of Test-Comp target two goals of test generation: "creating a test suite that covers a known bug in a given program" and "creating a test suite that covers all branches of a given program".

Thanks to the improvements Test-Comp brought, and six years after our original research [32], it is time to revisit the comparison: Model checkers vs. test generators—which tools are better at finding bugs in software?

✉ D. Beyer

[1] LMU Munich, Munich, Germany

We improve on the original comparison in multiple ways: (1) For the original work, we selected an array of test generators manually and configured them to the best of our knowledge. In this work, we base our comparison only on participants of the International Competition on Software Verification (SV-COMP) [16] and Test-Comp. All tool configuration is provided by the participating tool developers, and during the competition, developers got early access to prerun results to fix any shortcomings of their tools evident through the benchmark set.

(2) Originally, we executed our own, novel experiments. We do have high confidence in these results, but in our new work, we reuse the freely available competition data of SV-COMP 2023 and Test-Comp 2023. Using these results has the advantage that the data were peer-reviewed by the tool developers before publication.

Through these two adjustments we ensure that the used experimental data represent expert tool usage. They also guarantee that we configured everything correctly and that we select tools that support all of the major required language features.

(3) Originally, we counted that a model checker found a bug when the reported bug was confirmed by at least one witness validator [37]—which may solely rely on static analysis. In this work, we pay higher tribute to the actual execution of an error and separately consider whether a model checker bug report can be confirmed through program execution [39].

(4) Originally, we considered the bug-finding capabilities of model checkers and test generators but did not explicitly tune test generators toward finding a bug in the program. Our expectation is that many test generators are originally designed for traditional coverage measures like branch coverage or condition coverage and are not optimized to create a single test for an error location of interest. But since Test-Comp asks participants to create a test suite that covers a known bug, the Test-Comp test generators may be tuned toward bug finding. To check the effect of this, we compare the test suites generated by Test-Comp test generators for error coverage and the test suites generated for branch coverage with regards to their bug-finding capabilities.

(5) Furthermore, in the original work, we compared tools that market themselves as software model checkers with tools that market themselves as test generators and gave only a coarse overview on the techniques they used. Nowadays, many tools employ hybrid approaches with multiple different techniques. Many formal methods that are used in model checking can also be used for test generation [36, 112], and techniques originally designed for testing can be used as a part of model checking (for example, input fuzzing [56]). This means that a model checker and a test generator may use the same underlying analysis techniques. To account for that, we give more detail about the techniques the tools use.

We evaluate the following research questions:

RQ 1 Are test generators more effective in finding bugs than software model checkers?
RQ 2 Can the bug reports of software model checkers be validated through execution?
RQ 3 Are test generators that target errors more effective in finding bugs than test generators that target branch coverage?

To answer these questions, we use Test-Comp test generators and SV-COMP model checkers as representatives of their respective domains, with the original competition data. To the best of our knowledge, this is the first meta-analysis of the two international competitions SV-COMP and Test-Comp and the largest evaluation that compares the bug-finding capabilities of software model checkers with those of test generators.

Related work. The only large-scale comparisons of the tools considered in this work are the annual competitions SV-COMP [16] and Test-Comp [19], which we combine and inspect in detail in this work.

Next to these experimental evaluations, there are literature surveys on test generation for JavaScript [7], search-based testing [98], fuzzing [97], and symbolic execution [10, 48, 104]. There are also surveys on software-model-checking techniques [68, 86] and formal methods in a more general sense [13, 73], as well as the handbook on model checking [58].

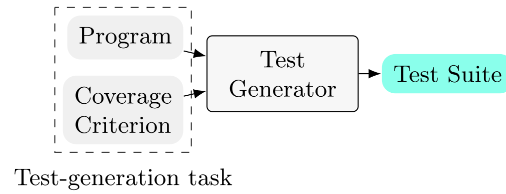


Fig. 1 Workflow of a Test-Comp test generator; a test generator produces a test suite for a program under test and a coverage criterion

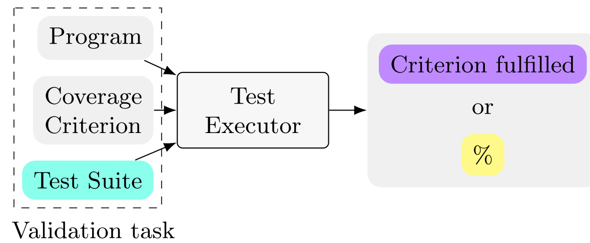


Fig. 2 Workflow of a test executor; a test executor computes whether (or to what percentage) a test suite fulfills a coverage criterion for a program

This work focuses on reachability bugs in a sequential, self-contained program, similar to a failing `assert` statement, and on tools and techniques aimed at finding such errors. Other applications of model checking and automated testing are, among many others, mutation testing [103] and the verification of concurrent programs [78], security properties [12], and hyperproperties [59].

2 Background

2.1 Testing

An input function in a program is any function that retrieves a value from the program environment; for example a system call. In our work, we use special functions `__VERIFIER_nondet_X`, which can return any input value of type `X`. For example, function `__VERIFIER_nondet_int()` returns an integer input value. A *test* vector $\langle v_0, \ldots, v_n \rangle$ is a sequence of n values. When $\langle v_0, \ldots, v_n \rangle$ is executed, the ith call to an input function is defined to return value v_i. A test suite is a set of test vectors. A test vector t covers a program operation op if the execution of t goes through op. A test suite covers a program operation op if any of its contained test vectors covers op.

A Test-Comp test generator (Fig. 1) [19] takes as input the program under test and a coverage criterion (e.g., cover a call to function `reach_error()`) and generates as output a test suite. The test executor (Fig. 2) then takes as input the program under test, the coverage criterion, and the generated test suite.

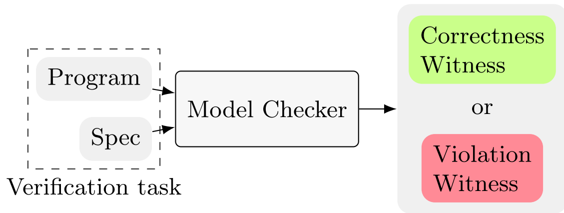


Fig. 3 Workflow of a model checker; a model checker produces a correctness witness if it claims that the program under verification fulfills the specification, or a violation witness if it claims that the program violates the specification

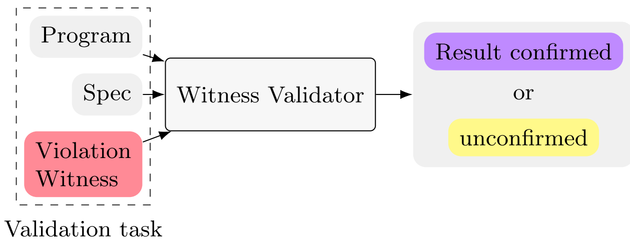


Fig. 4 Workflow of a witness validator (for result validation of a violation witness); a witness validator confirms the model checker verification result if it can reproduce the result with the help of the witness

```
1  unsigned char __VERIFIER_nondet_uchar();
2  void reach_error();
3
4  int main() {
5    unsigned char a =
6      __VERIFIER_nondet_uchar();
7    unsigned char b =
8      __VERIFIER_nondet_uchar();
9    unsigned char sum = a + b;
10   unsigned char mean = sum / 2;
11   if (mean < a / 2) {
12     reach_error();
13   }
14 }
```

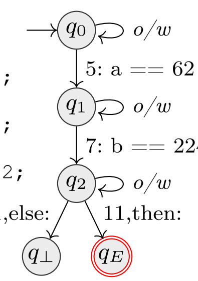


Fig. 5 Example program and violation-witness automaton (adapted from prior work [39])

It produces as output either that the coverage criterion is fulfilled or a percentage of how many coverage goals defined by the criterion are covered by the tests in the test suite.

2.2 Model checking

An SV-COMP model checker (Fig. 3) [16] takes as input a program and a specification and produces one of two outputs: If the program fulfills the specification, then a correctness witness [37, 41] is generated. If the program violates the specification, then a violation witness [38, 41] is generated.

2.3 Witness validation

Witness validation [41] aims to increase the trust in results of model checking. The idea is the following: A model checker (Fig. 3) analyzes a program with regards to a specification. As output, it not only produces a verification verdict *"property fulfilled"* or *"property not fulfilled"* but also a correctness witness or violation witness that helps to recreate the verification result. This witness is then given to a *witness validator* (Fig. 4). A witness validator takes the program under verification, the original specification, and the previously produced witness as input. It tries to reproduce the verification result with the help of the witness. If the witness validator is successful, then the result is confirmed, and confidence in the verification result increases.

In this work, we focus on bug-finding capabilities, so we only consider violation witnesses.

We describe violation witnesses as *violation-witness automata* (in version 1.0 [37], not yet version 2.0 [9]). A violation-witness automaton is a finite-state automaton. It contains at its transitions *source-code guards* e and *state-space guards* ψ to describe a subset of the program paths that contain the reported property violation. A source-code guard e is a program statement identified by its source-code line number. A source-code guard can also restrict the direction of program branchings; for example at `if` statements. It only allows the transition from one witness-automaton state to another if the currently considered program expression matches e and the specified program branch is entered (if specified). A state-space guard ψ is a predicate on the program state. It restricts the possible program states to those that fulfill ψ. Figure 5 shows an example program and a violation-witness automaton for the violated property `unreach-call`. Automaton label *o/w* describes a transition that is taken in all cases not covered by other transitions. This violation-witness automaton describes only the program state space that assigns `a = 62` and `b = 224`, which leads to an unsigned integer overflow and makes the program enter the **if** branch: The automaton stays in state q_0 until the assignment in line 5 is considered. It then transitions to q_1 and restricts the considered program states to those that fulfill $a == 62$ (after transitioning). When line 7 is reached, it restricts the considered program states to those that fulfill $b == 224$. When the **if** statement in line 11 is reached and the **if** branch is entered, the violation location is reached.

SV-COMP requires participants to output violation witnesses since SV-COMP 2015 [15]. It uses the XML-based GraphML exchange format [60]. Figure 6 shows an excerpt that represents the automaton displayed in Fig. 5.

Witness to test. Execution-based witness validation [39] takes a violation witness and tries to transform it into an executable test. If it succeeds, then the test is executed. If this test execution triggers the property violation, then the verification result is confirmed.

To generate the executable test, execution-based witness validation uses the source-code guards of the violation-witness automaton to map the corresponding state-space

```
<graph edgedefault="directed">
  <node id="q0">
   <data key="entry">true</data>
  </node>
  <node id="q1"/>
  <edge source="q0" target="q1">
   <data key="startline">5</data>
   <data key="assumption">a == (62U);</data>
   <data key="assumption.scope">main</data>
  </edge>
  <node id="q2"/>
  <edge source="q2" target="qE">
   <data key="startline">7</data>
   <data key="assumption">b == (224U);</data>
   <data key="assumption.scope">main</data>
  </edge>
  <node id="qE">
   <data key="violation">true</data>
  </node>
  <edge source="q2" target="qE">
   <data key="startline">11</data>
   <data key="control">condition-true</data>
  </edge>
  <node id="qBot">
   <data key="sink">true</data>
  </node>
  <edge source="q2" target="qBot">
   <data key="startline">11</data>
   <data key="control">condition-false</data>
  </edge>
</graph>
```

Fig. 6 Excerpt of the GraphML representation of the violation-witness automaton of Fig. 5

guards to the program code. If every call to an input function (`__VERIFIER_nondet_X`) is constrained to a unique assignment through a state-space guard (e.g., $a == 62$), then these unique assignments represent the test inputs; for example $\langle 62, 224 \rangle$. These inputs are then written to a test harness that allows for the execution of the test.

Because the result is confirmed by actual program execution, execution-based witness validation provides the same degree of confidence in the verification result as testing.

2.4 The benchmark collection SV-benchmarks

SV-Benchmarks [61] is the largest available collection of benchmark tasks for the evaluation of automated verification techniques for the language C. SV-Benchmarks contains *verification tasks* and *test-generation tasks*.

Verification task. A verification task of SV-Benchmarks consists of a program (C code) to verify and a program property to check. Program specifications are expressed in linear temporal logic and different properties exist: safety properties (e.g., error never reachable) and liveness properties (e.g., program always terminates). In this work, we only consider the safety property `unreach-call`, which specifies that no program execution may ever call the function `reach_error`.

Test-generation task. A test-generation task of SV-Benchmarks consists of a program (C code) to generate a test suite for and the coverage criterion which the test suite should fulfill. Coverage criteria are expressed as FQL [82], and, to date, two criteria exist: `coverage-error-call` asks for a test suite that covers at least one call to the function `reach_error` (signals a bug), and `coverage-branches` asks for a test suite that covers all branches of the program.

Categories. SV-Benchmarks groups benchmark tasks into categories. A detailed description of the categories is available online [110]. Table 1 gives an overview of the benchmark tasks with coverage criterion `coverage-error-call`, grouped by their categories. The table shows the category name, a description of the category, the number of benchmark tasks in that category, and a plot that illustrates the lines of program code per task in that category. Each plot shows on the x-axis the number of lines of code and on the y-axis the number of tasks in that category with the respective lines of code. In this work, we only consider these benchmark tasks.

3 Evaluation

3.1 Experiment setup

For all comparisons, we use the results obtained in SV-COMP and Test-Comp using the following setup: Experiments ran on machines with Intel Xeon E3-1230 v5 CPUs with 3.40 GHz, 8 cores, turbo boost disabled, and 33 GB of memory. For both competitions, each run of a verification task or test-generation task was limited to 900 s of CPU time, 15 GB of memory (RAM), and 8 CPU cores. Each violation-witness validation was limited to 90 s of CPU time, 7 GB of memory, and 2 CPU cores. Each test-suite validation was limited to 300 s of CPU time, 7 GB of memory, and 2 CPU cores. Resource limitation and measurement were performed by BenchExec [14, 40].

Note. On its web page [23], SV-COMP reports not only the score but also the run times of its participants. We refrain from reporting run time in this work because in Test-Comp, there is nothing wrong with fully using the available run time; the tools may continue generating tests until the time limit is hit, and they do.

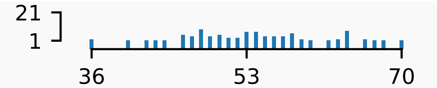
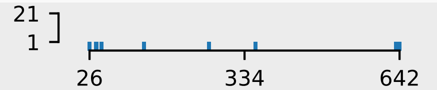
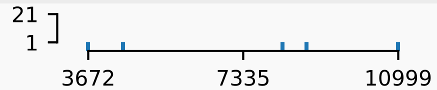
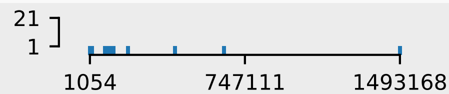
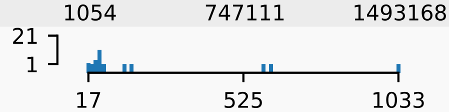
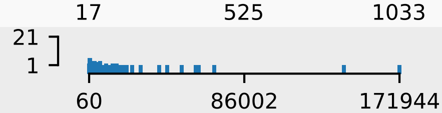
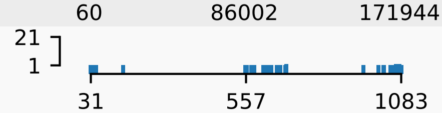
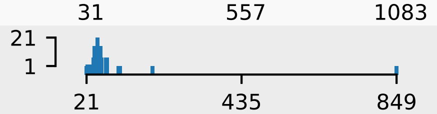
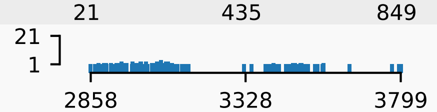
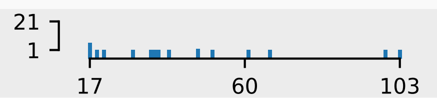
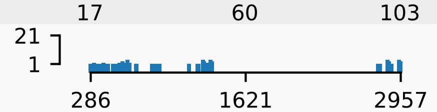
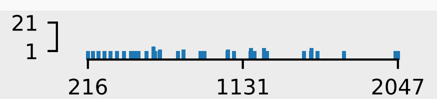
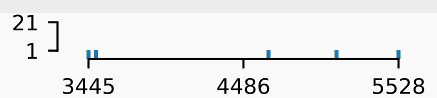
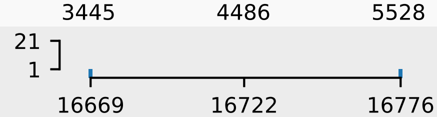
3.2 Benchmark tasks

We consider all benchmark tasks from the SV-Benchmarks repository with coverage criterion `coverage-error-call`.

3.3 Considered tools

We consider all 13 test generators that participated in Test-Comp 2023 and 31 software model checkers that participated in a subcategory of SV-COMP 2023 with checked

Table 1 Subcategories (14) of Test-Comp with coverage criterion `coverage-error-call`; each plot in the column "Lines of Code" illustrates the lines of program code per task in that category; each plot shows on the *x*-axis the number of lines of code, and on the *y*-axis the number of tasks in that category with the respective lines of code

Subcategory	Description	#Tasks	Lines of code
Arrays	Require treatment of arrays	90	
BitVectors	Require treatment of bit-operations	9	
ControlFlow	Program correctness depends mostly on the control-flow structure and integer variables	5	
ECA	Derived from event-condition-action systems	18	
Floats	Require treatment of floating-point arithmetics	32	
Hardware	Created from word-level hardware-model-checking benchmarks	494	
Heap	Require treatment of data structures on the heap, pointer aliases, and function pointers	47	
Loops	Require treatment of (potentially indeterminate) loops	130	
ProductLines	Represent "products" and "product simulators" that are derived using different configurations of product lines	169	
Recursive	Require treatment of recursive functions	20	
Sequentialized	Sequentialized concurrent programs that were derived from SystemC programs; the programs were transformed to pure C programs by incorporating the scheduler into the C code	98	
XCSP	Derived from constraint-programming benchmark tasks of combinatorial constrained problems	54	
BusyBox	Tasks from the software system BusyBox	5	
DeviceDriversLinux64	Tasks from the Linux Driver Verification project	2	

property `unreach-call` (excluding category ConcurrencySafety). Table 2 gives an overview on a selection of verification techniques used by each tool, based on data provided by the SV-COMP [16] and Test-Comp [19] competition reports. The reports do not list the identical set of techniques: if a report does not provide information on a technique, then this column is marked with ⊘ for the respective tools. The table groups the features on the *x*-axis in static techniques,

Table 2 Features used by Test-Comp and SV-COMP participants and their overall results in bug finding; if a competition report does not provide information on a technique, then this column is marked with ⊘ for the respective tools

	Participant	Static								Dyn.		Strategies										#Bugs found
		Bounded model checking	CEGAR	Explicit-value analysis	k-induction	Numeric interval analysis	Predicate abstraction	Shape analysis	Symbolic execution	Random execution	Evolutionary algorithms	ARG-based analysis	Bit-precise analysis	Floating-point arithmetics	Lazy abstraction	Interpolation	Automata-based analysis	Guidance by property	Targeted input generation	Algorithm selection	Portfolio	
Test-Comp	VeriFuzz [99, 100]	✓		✓	⊘	⊘		⊘		✓	✓	⊘	⊘	✓	⊘	⊘	⊘	✓				964
	FuSeBMC [5, 6]	✓			⊘	⊘		⊘		✓	✓	⊘	⊘	✓	⊘	⊘	⊘	✓	✓		✓	939
	FuSeBMC_IA [4]	✓			⊘	⊘		⊘		✓	✓	⊘	⊘	✓	⊘	⊘	⊘	✓	✓		✓	931
	CoVeriTest [29, 85]		✓	✓	⊘	⊘	✓	⊘				⊘	⊘	✓	⊘	⊘	⊘				✓	564
	Klee [47, 49]				⊘	⊘		⊘	✓			⊘	⊘	✓	⊘	⊘	⊘		✓			541
	Symbiotic [52, 53]				⊘	⊘		⊘	✓			⊘	⊘	✓	⊘	⊘	⊘	✓	✓		✓	510
	TracerX [83, 84]	✓			⊘	⊘		⊘	✓			⊘	⊘	✓	⊘	⊘	⊘		✓			420
	HybridTiger [46, 107]		✓	✓	⊘	⊘	✓	⊘				⊘	⊘	✓	⊘	⊘	⊘					397
	WASP-C [115]				⊘	⊘		⊘	✓	✓		⊘	⊘	✓	⊘	⊘	⊘					393
	Esbmc-kind [71, 72]	✓			⊘	⊘		⊘				⊘	⊘		⊘	⊘	⊘	✓				352
	PRTest [32, 93]				⊘	⊘		⊘		✓		⊘	⊘	✓	⊘	⊘	⊘					293
	Legion/SymCC [92]			✓	⊘	⊘		⊘	✓	✓		⊘	⊘	✓	⊘	⊘	⊘	✓	✓			281
	Legion [94, 95]			✓	⊘	⊘		⊘	✓	✓		⊘	⊘	✓	⊘	⊘	⊘	✓	✓			108
SV-COMP	PeSCo [105, 106]	✓	✓	✓	✓	✓	✓	✓		⊘		✓	✓	⊘	✓	✓		✓	⊘	✓	✓	667
	CPAchecker [31, 63]	✓	✓	✓	✓	✓	✓	✓	✓	⊘		✓	✓	⊘	✓	✓		✓	⊘	✓	✓	665
	Esbmc-kind [71, 72]	✓			✓	✓				⊘			✓	⊘				✓	⊘			660
	VeriAbsL [65]	✓	✓	✓	✓	✓	✓			⊘	✓			⊘				✓	⊘	✓	✓	645
	Graves-CPA [91]									⊘				⊘					⊘			643
	VeriAbs [3, 64]	✓	✓	✓	✓	✓	✓			⊘	✓			⊘				✓	⊘	✓	✓	639
	Bubaak [51]								✓	⊘			✓	⊘				✓	⊘			635
	Cbmc [57, 89]	✓								⊘			✓	⊘				✓	⊘			626
	VeriFuzz [56, 99]	✓				✓				⊘	✓			⊘				✓	⊘			615
	CVT-ParPort [30, 42]	✓	✓	✓	✓	✓	✓	✓	✓	⊘		✓	✓	⊘	✓	✓		✓	⊘	✓	✓	591
	Symbiotic [52, 54]				✓	✓		✓	✓	⊘			✓	⊘				✓	⊘		✓	559
	CVT-AlgoSel [30, 42]	✓	✓	✓	✓	✓	✓	✓	✓	⊘		✓	✓	⊘	✓	✓	✓	✓	⊘	✓	✓	468
	UAutomizer [80, 81]		✓				✓			⊘			✓	⊘	✓	✓	✓	✓	⊘	✓	✓	311
	Divine [11, 90]			✓					✓	⊘			✓	⊘				✓	⊘	✓	✓	299
	UTaipan [66, 77]		✓	✓		✓	✓			⊘			✓	⊘	✓	✓	✓	✓	⊘	✓	✓	294
	Pinaka [55]	✓							✓	⊘			✓	⊘				✓	⊘			272
	gazer-theta [1, 79]	✓	✓	✓			✓			⊘		✓	✓	⊘	✓	✓		✓	⊘		✓	255
	2ls [44, 96]	✓			✓	✓		✓		⊘			✓	⊘				✓	⊘			213
	UKojak [69, 102]		✓				✓			⊘			✓	⊘	✓	✓		✓	⊘			189
	Crux [67, 109]								✓	⊘				⊘				✓	⊘			176

Table 2 (*Continued*)

Participant	Static: Bounded model checking	Static: CEGAR	Static: Explicit-value analysis	Static: k-induction	Static: Numeric interval analysis	Static: Predicate abstraction	Static: Shape analysis	Static: Symbolic execution	Dyn.: Random execution	Dyn.: Evolutionary algorithms	Strategies: ARG-based analysis	Strategies: Bit-precise analysis	Strategies: Floating-point arithmetics	Strategies: Lazy abstraction	Strategies: Interpolation	Strategies: Automata-based analysis	Strategies: Guidance by property	Strategies: Targeted input generation	Strategies: Algorithm selection	Strategies: Portfolio	#Bugs found
KORN [70]			✓			✓		✓	⊘				⊘				✓	⊘		✓	121
THETA [2, 111]		✓	✓			✓			⊘		✓	✓	⊘		✓		✓	⊘	✓	✓	116
BRICK [45]	✓	✓			✓			✓	⊘				⊘				✓	⊘			99
GRAVES-PAR [76]									⊘				⊘					⊘			93
GDART-LLVM [74]								✓	⊘			✓	⊘					⊘			1

Table 3 Data that we use from the competitions

Artifact	DOI
Benchmark collection	10.5281/zenodo.7627783
SV-COMP results	10.5281/zenodo.7627787
Test-Comp results	10.5281/zenodo.7701122
Test-Comp test suites	10.5281/zenodo.7701126
Test-suite validator	10.5281/zenodo.7701118

dynamic techniques, and strategies in verification that can be used with both static and dynamic techniques. The tools are grouped on the y-axis by SV-COMP and Test-Comp participation. Within each group, the entries are sorted by the number of found bugs over all benchmark tasks. We omit tools that did not find a single confirmed bug in the considered verification tasks: CPA-BAM-BNB [8, 114], CPA-BAM-SMG, FRAMA-C-SV [35, 62], GOBLINT [108, 113], INFER-SV [50, 88], and MOPSA [87, 101].

The table shows that most test generators that participated in Test-Comp 2023 use hybrid approaches: they employ both static and dynamic analysis techniques.

Table 3 shows the external data from the competitions we used for our study.

3.4 Expanding the study

To add new tools to the tool comparison, developers can submit their tool to the next iterations of SV-COMP [28] and Test-Comp [27]. For private experiments, the benchmarking configuration is available online and described on the competition websites of SV-COMP [24] and Test-Comp [26]. Competition results can be analyzed with scripts from our reproduction artifact [34].

3.5 Experimental results

RQ 1. Are test generators more effective in finding bugs than software model checkers? We use the original results data of SV-COMP 2023 [17] and Test-Comp 2023 [18]. To make the two data sets comparable, we map all results for test-generation tasks in the Test-Comp data to results for a verification task with property `unreach-call`: Each successful test generation for coverage criterion `coverage-error-call` also produces a valid counterexample for `unreach-call`. This means that if a test generator successfully generates a test suite that fulfills criterion `coverage-error-call`, then it also shows that `unreach-call` is violated. For both SV-COMP and Test-Comp data, we only consider a bug "found" if it is confirmed by the competition through successful violation-witness validation or test execution.

We report the highest bug-finding capability each tool exhibits in its respective competition. The tool TRACERX only produces test suites for `coverage-branches`, and for LEGION/SYMCC, the test suites generated for `coverage-branches` cover more bugs than the test suites generated for `coverage-error-call` (cf. RQ 3). For these tools, we always consider the test suites they generated for `coverage-branches`.

Table 2 (right column) shows the overall number of tasks for which a bug was found by the resp. tool. In contrast to our original study [32], two test generators VERIFUZZ [100] (964/1173 bugs found) and FUSEBMC [5] (939/1173 bugs found) perform significantly better than the best model checker PESCO [105, 106] (667/1173 bugs found). Both VERIFUZZ and FUSEBMC use a combination of bounded model checking [43] (a static technique) and fuzzing [75] (a dynamic technique).

Two notes. (1) Some of the model checkers listed in Table 2 are specialized tools that (a) participate only in selected categories of SV-COMP or (b) focus on program proofs, not bug hunting. For these reasons, a low number of found bugs gives no indication about the tool's quality. For example, GDART-LLVM has the lowest overall number of found bugs, but it only participates in category BitVectors. The best three model checkers, PESCO, CPACHECKER, and ESBMC-KIND, participate in all relevant categories. (2) The reported numbers do not match the Test-Comp overall scores reported on the official results page [25] because Test-Comp performs normalization over each category number of tasks. We do not perform normalization but report the sum of all found bugs over all categories.

The tools ESBMC-KIND, SYMBIOTIC, and VERIFUZZ participated in both SV-COMP and Test-Comp. If not clear from the context, we superscript their names with the competition in which the result was received (for example, VERIFUZZ$^{\text{SV-COMP}}$ or SYMBIOTIC$^{\text{Test-Comp}}$). If the results are equal for both configurations, then we write VERIFUZZ$^{\text{Both}}$.

Table 4 displays the results of the selected tools per category. For each category, the table lists data for the three best test generators and three best model checkers that found at least one bug in that category (four tools each for category Overall). If there is a draw, then all tools with the same number of found bugs and with the same number of bugs confirmed through execution (cf. RQ 2) are displayed. To ease the differentiation between the two groups, we prefix each test generator with T and each model checker with M. The table lists the total tasks in the respective category, the number of confirmed bugs that the respective tool found, as well as the number of bugs that the respective tool found and that were confirmed by actual program execution. We omit the category DeviceDriversLinux64 because no tool was able to find a bug in it.

The table shows that for bug finding, individual test generators perform either better or as good as individual model checkers in all categories but Heap and XCSP. A clear divide between test generators and model checkers exists in four categories: In Arrays, the best test generator of that category, FUSEBMC, finds a bug in 90 tasks, whereas the best model checker of that category, VERIABSL, finds a bug in only 81 tasks. In Hardware, VERIFUZZ finds a bug in 319 tasks, whereas GRAVES-CPA finds a bug in only 147 tasks. In Loops, FUSEBMC finds a bug in 128 tasks, whereas VERIABS finds a bug in only 112 tasks. In Sequentialized, VERIFUZZ finds a bug in 95 tasks, whereas PESCO finds a bugs in only 86 tasks.

The presented data answers our first research question with "yes": At the current state-of-the-art for C, test generators perform significantly better in bug hunting than model checkers.

Table 4 Results of the tools listed in Table 2 for each category; only the best test generators (T) and model checkers (M) of each category are listed

		Total tasks	#Bugs found	#Bugs confirmed by execution
Arrays				
T	FUSEBMC	90	**90**	90
T	FUSEBMC_IA	90	88	88
T	VERIFUZZ$^{\text{Test-Comp}}$	90	88	88
M	VERIABSL	90	81	76
M	VERIABS	90	80	66
M	BUBAAK	90	74	74
BitVectors				
T	FUSEBMC	9	**9**	9
T	FUSEBMC_IA	9	**9**	9
T|M	VERIFUZZ$^{\text{Both}}$	9	**9**	9
M	SYMBIOTIC$^{\text{SV-COMP}}$	9	8	8
M	ESBMC-KIND$^{\text{SV-COMP}}$	9	8	6
M	GRAVES-CPA	9	8	6
ControlFlow				
T	FUSEBMC	5	**5**	5
T	FUSEBMC_IA	5	**5**	5
T|M	SYMBIOTIC$^{\text{Both}}$	5	**5**	5
M	BUBAAK	5	4	4
T|M	(VERIFUZZ)$^{\text{Both}}$	5	4	4
T	KLEE	5	4	4
ECA				
T	VERIFUZZ$^{\text{SV-COMP}}$	18	**15**	13
T	KLEE	18	14	14
M	BUBAAK	18	14	12
M	SYMBIOTIC$^{\text{Test-Comp}}$	18	13	13
M	PESCO	18	13	12
T	FUSEBMC	18	12	12
Floats				
T	FUSEBMC	32	**32**	32
T	FUSEBMC_IA	32	31	31
T	VERIFUZZ$^{\text{Test-Comp}}$	32	31	31
M	BRICK	32	30	29
M	CVT-PARPORT	32	30	24
M	CPACHECKER	32	30	21
Hardware				
T	VERIFUZZ$^{\text{Test-Comp}}$	494	**319**	319
T	FUSEBMC	494	288	288
T	FUSEBMC_IA	494	288	288
M	GRAVES-CPA	494	147	102
M	CPACHECKER	494	127	70
M	PESCO	494	109	61

Table 4 (*Continued*)

		Total tasks	#Bugs found	#Bugs confirmed by execution
Heap				
M	CBMC	47	**47**	43
M	VERIABS	47	47	33
M	BUBAAK	47	46	44
T	FUSEBMC	47	45	45
T	FUSEBMC_IA	47	45	45
T	KLEE	47	45	45
T|M	VERIFUZZ[Both]	47	45	45
Loops				
T	FUSEBMC	130	**128**	128
T	FUSEBMC_IA	130	127	127
T	VERIFUZZ[Test-Comp]	130	123	123
M	VERIABS	130	112	103
M	VERIABSL	130	100	86
M	KORN	130	98	97
ProductLines				
T	FUSEBMC	169	**169**	169
T	FUSEBMC_IA	169	**169**	169
T	KLEE	169	**169**	169
T|M	VERIFUZZ[Both]	169	**169**	169
M	BUBAAK	169	**169**	169
M	VERIABSL	169	**169**	169
Recursive				
T	FUSEBMC	20	**19**	19
T	FUSEBMC_IA	20	**19**	19
M	CBMC	20	**19**	19
M	CVT-PARPORT	20	**19**	19
M	GRAVES-CPA	20	19	17
T	VERIFUZZ[Test-Comp]	20	18	18
Sequentialized				
T	VERIFUZZ[Test-Comp]	98	**95**	95
T	FUSEBMC	98	94	94
T	FUSEBMC_IA	98	92	92
M	PESCO	98	86	86
M	CVT-PARPORT	98	86	32
M	CBMC	98	85	29
XCSP				
M	CBMC	54	**50**	50
M	CVT-ALGOSEL	54	49	49
T|M	VERIFUZZ[Both]	54	49	49
T	WASP-C	54	49	49
T|M	ESBMC-KIND[Both]	54	48	48
T	FUSEBMC	54	47	47

Table 4 (*Continued*)

		Total tasks	#Bugs found	#Bugs confirmed by execution
BusyBox				
T	FUSEBMC	5	**1**	1
T	KLEE	5	**1**	1
M	PESCO	5	1	0
Overall				
T	VERIFUZZ[Test-Comp]	1173	**964**	964
T	FUSEBMC	1173	939	939
T	FUSEBMC_IA	1173	931	931
M	PESCO	1173	667	475
M	CPACHECKER	1173	665	458
M	ESBMC-KIND[SV-COMP]	1173	660	529
M	VERIABSL	1173	645	543
T	COVERITEST	1173	564	564

Table 5 Number of bugs found by the best tool of each category, the union of all test generators (T), the union of all model checkers (M), and all tools

Category	Best tool	All T	All M	All tools
Arrays	90	87	90	90
BitVectors	9	9	9	9
ControlFlow	5	5	5	5
ECA	15	15	14	**17**
Floats	32	32	32	32
Hardware	319	340	175	**342**
Heap	47	45	47	47
Loops	128	128	127	128
ProductLines	169	169	169	169
Recursive	19	19	**20**	20
Sequentialized	95	95	90	95
XCSP	50	**51**	50	51
BusyBox	1	1	1	**2**

In our previous research study [32], the different tools complemented each other well, so that the combination of multiple tools yielded significant improvements in the number of bugs found. This is not true for the current results: Table 5 shows for each benchmark category the number of bugs found by the best tool in that category, the union of distinct bugs found by all test generators together (All T), the union of distinct bugs found by all model checkers together (All M), and the union of all considered tools (All Tools). The table shows that the unions only yield an improvement in 5 of the 13 categories and that these improvements are also small. We explain this with the fact that, in contrast to the previous study, almost all currently considered tools already combine multiple approaches internally (cf. Table 2), rendering further external combinations effectless.

RQ 2. Can the bug reports of software model checkers be validated through execution? Since a failing program execution provides the highest level of confidence in a verification result, we separately check how many of the confirmed verification results were confirmed not only by a third-party tool, but also by actual program execution.

For this, we use the SV-COMP validation results of the two execution-based witness validators CPA-WITNESS2TEST and FSHELL-WITNESS2TEST. Table 4 shows in its last columns the number of found bugs that are confirmed through program execution. It is visible that the confirmation rate can be very high; for example, for BRICK in category Floats (29 of 30), for CBMC in categories Heap (43 of 47), Recursive (19 of 19) and XCSP (50 of 50), or for PESCO in category Sequentialized (86 of 86). On the other hand, the confirmation rate can also be very low, even for model checkers that perform well otherwise and in categories that other model checkers perform well in: CBMC gets only 29 of 85 results confirmed through execution in category Sequentialized, and PESCO gets only 61 of 109 results confirmed in category Hardware. This hints to bug reports (in the form of violation witnesses) that miss input values.

Thus our answer to the second research question: The data show that the execution-based validation of verification results is feasible and works well to provide a similar level of confidence in the result of model checkers as in test generators. But at the current state-of-the-art, model checkers have to produce more precise violation witnesses to offer the same level of confidence as test generators.

RQ 3. Are test generators that target errors more effective in finding bugs than test generators that target branch coverage? To answer our last research question, we consider the test suites [22] that each test generator generated for coverage criterion `coverage-branches` in Test-Comp 2023. We check how well these test suites perform for finding bugs, compared to the test suites that testers specifically generated for bug-finding: We give each test suite generated for `coverage-branches` to the test executor of Test-Comp 2023, TESTCOV [33], but with target measure `coverage-error-call`. The results over all common categories are presented in Table 6.[1]

It is visible that 6 testers produce significantly better test suites for criterion `coverage-error-call` when told to do so: FUSEBMC, VERIFUZZ, FUSEBMC_IA, SYMBIOTIC, and, with the most notable difference, KLEE. This shows that they adjust their behavior based on the coverage criterion provided to them. The other tools only show very little difference between the two generated test suites or did not provide test suites for both coverage criteria. It is notable that the five best-performing testers all adjust their behavior based on the coverage criterion.

Table 6 Bug-finding capabilities of generated test suites that are targeted at either `coverage-error-call` or `coverage-branches`; the results exclude category Hardware because it is not part of the Test-Comp 2023 track on branch coverage

Tools	Total tasks	#Bugs found `error-call`	#Bugs found `branches`
FUSEBMC	679	**651**	594
VERIFUZZ	679	**645**	611
FUSEBMC_IA	679	**643**	594
KLEE	679	**541**	285
COVERITEST	679	**479**	476
SYMBIOTIC	679	**476**	456
TRACERX	679	-	420
HYBRIDTIGER	679	**362**	281
WASP-C	679	354	**355**
LEGION/SYMCC	679	279	**281**
ESBMC-KIND	679	352	-
PRTEST	679	236	236
LEGION	679	**108**	107

This answers our third research question with "yes": Testers that actively target errors are more effective in creating test suites for error coverage.

3.6 Threats to validity

Internal validity. We are confident in the internal validity of our analysis. We use the official SV-COMP 2023 and Test-Comp 2023 data. Both competitions pay highest priority to precise measurements and reproducibility. For validating test suites with `coverage-error-call` which were generated for `coverage-branches`, we had to perform own experiments. For these, we used the official competitions' infrastructure to ensure correctness of results. Both our setup and the produced data are publicly available [34] for inspection.

External validity. We use the largest available benchmark set with well-defined C programs for testing. Still, this benchmark set may not represent the full diversity of real-world C programs. Similarly, because tools know the SV-COMP and Test-Comp benchmark tasks before the competition runs, tools that participate in SV-COMP and Test-Comp may be tuned to the competitions' benchmark set and perform worse on real-world projects.

The application domain we can consider is limited: We consider testing of sequential, self-sufficient C programs with a simple reachability specification, similar to `assert` state-

[1] This excludes category Hardware, which only exists in the track for `coverage-error-call`.

ments (cf. Table 1). This means that the presented results may ignore program features and some applications of testing, like string handling, object-oriented programming, concurrency, or database queries.

Similarly, specific applications of verification, for example the verification of network protocols or static application-security testing, are not considered.

We only consider programs with at least one existing bug. We do not measure how good the generated test suites are for detecting bugs that are newly introduced in the future.

We also do not differentiate between a single found bug and multiple found bugs. But a test suite that detects multiple bugs in a program may be considered better than a test suite that only detects a single bug. We consider both options orthogonal research questions.

We only consider tools that participate in either SV-COMP 2023 or Test-Comp 2023. This covers the latest state-of-the-art for verification of C programs. There may still be model checkers or test generators that did not participate in the last iterations of SV-COMP or Test-Comp and that perform significantly better. In addition, the comparison of test generators and model checkers may differ in areas of application other than those considered.

Construct validity. We designed our experiments to assess whether test generators or model checkers find more bugs in given programs. To quantify the quality of the tools, we use the number of bugs found, which is the main ingredient of the community-agreed scoring schemas that the competitions use (considering the category FalsificationOverall in SV-COMP and category Cover-Error in Test-Comp). Instead of normalization as used in the competitions, we explicitly report the results per category in Table 4.

4 Conclusions

We performed a thorough comparison of the bug-finding capabilities for C programs of all SV-COMP 2023 and Test-Comp 2023 participants. This comparison shows that, although state-of-the-art test generators and model checkers are highly competitive, the best considered test generators outperform the best considered model checkers in bug finding. Notably, the best test generators do not limit themselves to dynamic techniques but also use static-analysis techniques and formal methods. FuSeBMC [5] and VeriFuzz [100] use a combination of bounded model checking [43] and fuzzing [75].

Funding Statement Open Access funding enabled and organized by Projekt DEAL. This work was funded by the Deutsche Forschungsgesellschaft (DFG) — 418257054 (Coop), the LMU PostDoc support fund, and the Free State of Bavaria.

Data-Availability Statement The analysis and all experimental data are archived and available at Zenodo [34]. We used the following existing data for our study: the benchmark collection that was used by both competitions [20], the SV-COMP results [17], the Test-Comp results [18] and test suites [22], and the test-suite validator TestCov [21] from Test-Comp. See Table 3.

References

1. Ádám, Z., Sallai, G., Hajdu, Á.: Gazer-Theta: LLVM-based verifier portfolio with BMC/CEGAR (competition contribution). In: Proc. TACAS (2). LNCS, vol. 12652, pp. 433–437. Springer, Berlin (2021). https://doi.org/10.1007/978-3-030-72013-1_27
2. Ádám, Z., Bajczi, L., Dobos-Kovács, M., Hajdu, A., Molnár, V.: Theta: portfolio of cegar-based analyses with dynamic algorithm selection (competition contribution). In: Proc. TACAS (2). LNCS, vol. 13244, pp. 474–478. Springer, Berlin (2022). https://doi.org/10.1007/978-3-030-99527-0_34
3. Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: VeriAbs: verification by abstraction and test generation. In: Proc. ASE, pp. 1138–1141. IEEE (2019). https://doi.org/10.1109/ASE.2019.00121
4. Aldughaim, M., Alshmrany, K.M., Gadelha, M.R., de Freitas, R., Cordeiro, L.C.: FuSeBMC_IA: interval analysis and methods for test-case generation (competition contribution). In: Proc. FASE. LNCS, vol. 13991, pp. 324–329. Springer, Berlin (2023). https://doi.org/10.1007/978-3-031-30826-0_18
5. 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. Springer, Berlin (2021). https://doi.org/10.1007/978-3-030-79379-1_6
6. Alshmrany, K., Aldughaim, M., Cordeiro, L., Bhayat, A.: FuSeBMC v. 4: smart seed generation for hybrid fuzzing (competition contribution). In: Proc. FASE. LNCS, vol. 13241, pp. 336–340. Springer, Berlin (2022). https://doi.org/10.1007/978-3-030-99429-7_19
7. Andreasen, E., Gong, L., Møller, A., Pradel, M., Selakovic, M., Sen, K., Staicu, C.: A survey of dynamic analysis and test generation for JavaScript. ACM Comput. Surv. **50**(5), 66:1–66:36 (2017). https://doi.org/10.1145/3106739
8. Andrianov, P., Friedberger, K., Mandrykin, M.U., Mutilin, V.S., Volkov, A.: CPA-BAM-BnB: block-abstraction memoization and region-based memory models for predicate abstractions (competition contribution). In: Proc. TACAS. LNCS, vol. 10206, pp. 355–359. Springer, Berlin (2017). https://doi.org/10.1007/978-3-662-54580-5_22
9. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. In: Proc. SPIN. LNCS, vol. 14624. Springer, Berlin (2024). https://doi.org/10.1007/978-3-031-66149-5_11
10. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic-execution techniques. ACM Comput. Surv. **51**(3), 50:1–50:39 (2018). https://doi.org/10.1145/3182657

11. Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., Štill, V.: Model checking of C and C++ with Divine 4. In: Proc. ATVA. LNCS, vol. 10482, pp. 201–207. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-68167-2_14
12. Basin, D.A., Cremers, C., Meadows, C.A.: Model checking security protocols. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 727–762. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-10575-8_22
13. Beckert, B., Hähnle, R.: Reasoning and verification: state of the art and current trends. IEEE Intell. Syst. **29**(1), 20–29 (2014). https://doi.org/10.1109/MIS.2014.3
14. BenchExec: A framework for reliable benchmarking and resource measurement. https://github.com/sosy-lab/benchexec. Accessed: 2024–10–31
15. Beyer, D.: Software verification and verifiable witnesses (report on SV-COMP 2015). In: Proc. TACAS. LNCS, vol. 9035, pp. 401–416. Springer, Berlin (2015). https://doi.org/10.1007/978-3-662-46681-0_31
16. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). LNCS, vol. 13994, pp. 495–522. Springer, Berlin (2023). https://doi.org/10.1007/978-3-031-30820-8_29
17. Beyer, D.: Results of the 12th Intl. Competition on Software Verification (SV-COMP 2023). Zenodo (2023). https://doi.org/10.5281/zenodo.7627787
18. Beyer, D.: Results of the 5th Intl. Competition on Software Testing (Test-Comp 2023). Zenodo (2023). https://doi.org/10.5281/zenodo.7701122
19. Beyer, D.: Software testing: 5th comparative evaluation: Test-Comp 2023. In: Proc. FASE. LNCS, vol. 13991, pp. 309–323. Springer, Berlin (2023). https://doi.org/10.1007/978-3-031-30826-0_17
20. Beyer, D.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2023 and Test-Comp 2023). Zenodo (2023). https://doi.org/10.5281/zenodo.7627783
21. Beyer, D.: Test-suite generators and validator of the 5th Intl. Competition on Software Testing (Test-Comp 2023). Zenodo (2023). https://doi.org/10.5281/zenodo.7701118
22. Beyer, D.: Test suites from test-generation tools (Test-Comp 2023). Zenodo (2023). https://doi.org/10.5281/zenodo.7701126
23. Beyer, D.: 12th Intl. Competition on Software Verification (SV-COMP 2023): Results of the Competition. https://sv-comp.sosy-lab.org/2023/results/results-verified/. Accessed: 2024–10–31
24. Beyer, D.: 12th Intl. Competition on Software Verification (SV-COMP 2023): Submission. https://sv-comp.sosy-lab.org/2023/submission.php. Accessed: 2024–10–31
25. Beyer, D.: 5th Intl. Competition on Software Testing (Test-Comp 2023): Results of the Competition. https://test-comp.sosy-lab.org/2023/results/results-verified/. Accessed: 2024–10–31
26. Beyer, D.: 5th Intl. Competition on Software Testing (Test-Comp 2023): Submission. https://test-comp.sosy-lab.org/2023/submission.php. Accessed: 2024–10–31
27. Beyer, D.: Intl. Competition on Software Testing (Test-Comp). https://test-comp.sosy-lab.org/. Accessed: 2024–10–31
28. Beyer, D.: Intl. Competition on Software Verification (SV-COMP). https://sv-comp.sosy-lab.org/. Accessed: 2024–10—31
29. Beyer, D., Jakobs, M.C.: Cooperative verifier-based testing with CoVeriTest. Int. J. Softw. Tools Technol. Transf. **23**(3), 313–333 (2021). https://doi.org/10.1007/s10009-020-00587-8
30. Beyer, D., Kanav, S.: CoVeriTeam: on-demand composition of cooperative verification systems. In: Proc. TACAS. LNCS, vol. 13243, pp. 561–579. Springer, Berlin (2022). https://doi.org/10.1007/978-3-030-99524-9_31
31. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Proc. CAV. LNCS, vol. 6806, pp. 184–190. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-22110-1_16
32. Beyer, D., Lemberger, T.: Software verification: testing vs. model checking. In: Proc. HVC. LNCS, vol. 10629, pp. 99–114. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-70389-3_7
33. Beyer, D., Lemberger, T.: TestCov: robust test-suite execution and coverage measurement. In: Proc. ASE, pp. 1074–1077. IEEE (2019). https://doi.org/10.1109/ASE.2019.00105
34. Beyer, D., Lemberger, T.: Reproduction Package for STTT Article "Six Years Later: Testing vs. Model Checking". Zenodo (2023). https://doi.org/10.5281/zenodo.10232648
35. Beyer, D., Spiessl, M.: The static analyzer Frama-C in SV-COMP (competition contribution). In: Proc. TACAS (2). LNCS, vol. 13244, pp. 429–434. Springer, Berlin (2022). https://doi.org/10.1007/978-3-030-99527-0_26
36. 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
37. 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, New York (2015). https://doi.org/10.1145/2786805.2786867
38. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: exchanging verification results between verifiers. In: Proc. FSE, pp. 326–337. ACM, New York (2016). https://doi.org/10.1145/2950290.2950351
39. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: execution-based validation of verification results. In: Proc. TAP. LNCS, vol. 10889, pp. 3–23. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-92994-1_1
40. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
41. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). https://doi.org/10.1145/3477579
42. Beyer, D., Kanav, S., Richter, C.: Construction of verifier combinations based on off-the-shelf verifiers. In: Proc. FASE, pp. 49–70. Springer, Berlin (2022). https://doi.org/10.1007/978-3-030-99429-7_3
43. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. LNCS, vol. 1579, pp. 193–207. Springer, Berlin (1999). https://doi.org/10.1007/3-540-49059-0_14
44. Brain, M., Joshi, S., Kröning, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Proc. SAS. LNCS, vol. 9291, pp. 145–161. Springer, Berlin (2015). https://doi.org/10.1007/978-3-662-48288-9_9
45. Bu, L., Xie, Z., Lyu, L., Li, Y., Guo, X., Zhao, J., Li, X.: Brick: path enumeration-based bounded reachability checking of C programs (competition contribution). In: Proc. TACAS (2). LNCS, vol. 13244, pp. 408–412. Springer, Berlin (2022). https://doi.org/10.1007/978-3-030-99527-0_22
46. Bürdek, J., Lochau, M., Bauregger, S., Holzer, A., von Rhein, A., Apel, S., Beyer, D.: Facilitating reuse in multi-goal test-suite generation for software product lines. In: Proc. FASE. LNCS, vol. 9033, pp. 84–99. Springer, Berlin (2015). https://doi.org/10.1007/978-3-662-46675-9_6
47. Cadar, C., Nowack, M.: Klee symbolic execution engine in 2019 (competition contribution). Int. J. Softw. Tools Technol. Transf. **23**(6), 867–870 (2021). https://doi.org/10.1007/s10009-020-00570-3

48. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013). https://doi.org/10.1145/2408776.2408795
49. 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)
50. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. LNCS, vol. 9058, pp. 3–11. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-17524-9_1
51. Chalupa, M., Henzinger, T.: Bubaak: runtime monitoring of program verifiers (competition contribution). In: Proc. TACAS (2). LNCS, vol. 13994, pp. 535–540. Springer, Berlin (2023). https://doi.org/10.1007/978-3-031-30820-8_32
52. Chalupa, M., Strejček, J., Vitovská, M.: Joint forces for memory safety checking. In: Proc. SPIN, pp. 115–132. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-94111-0_7
53. Chalupa, M., Novák, J., Strejček, J.: Symbiotic 8: parallel and targeted test generation (competition contribution). In: Proc. FASE. LNCS, vol. 12649, pp. 368–372. Springer, Berlin (2021). https://doi.org/10.1007/978-3-030-71500-7_20
54. Chalupa, M., Řechtáčková, A., Mihalkovič, V., Zaoral, L., Strejček, J.: Symbiotic 9: string analysis and backward symbolic execution with loop folding (competition contribution). In: Proc. TACAS (2). LNCS, vol. 13244, pp. 462–467. Springer, Berlin (2022). https://doi.org/10.1007/978-3-030-99527-0_32
55. Chaudhary, E., Joshi, S.: Pinaka: symbolic execution meets incremental solving (competition contribution). In: Proc. TACAS (3). LNCS, vol. 11429, pp. 234–238. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-17502-3_20
56. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: VeriFuzz: program-aware fuzzing (competition contribution). In: Proc. TACAS (3). LNCS, vol. 11429, pp. 244–249. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-17502-3_22
57. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. LNCS, vol. 2988, pp. 168–176. Springer, Berlin (2004). https://doi.org/10.1007/978-3-540-24730-2_15
58. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of Model Checking. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-10575-8
59. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010). https://doi.org/10.3233/JCS-2009-0393
60. Collection of verification tasks. https://gitlab.com/sosy-lab/benchmarking/sv-witnesses. Accessed: 2024–10–31
61. Collection of verification tasks. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks. Accessed: 2024–10–31
62. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Proc. SEFM, pp. 233–247. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-33826-7_16
63. Dangl, M., Löwe, S., Wendler, P.: CPAchecker with support for recursive programs and floating-point arithmetic (competition contribution). In: Proc. TACAS. LNCS, vol. 9035, pp. 423–425. Springer, Berlin (2015). https://doi.org/10.1007/978-3-662-46681-0_34
64. Darke, P., Agrawal, S., Venkatesh, R.: VeriAbs: a tool for scalable verification by abstraction (competition contribution). In: Proc. TACAS (2). LNCS, vol. 12652, pp. 458–462. Springer, Berlin (2021). https://doi.org/10.1007/978-3-030-72013-1_32
65. Darke, P., Chimdyalwar, B., Agrawal, S., Venkatesh, R., Chakraborty, S., Kumar, S.: VeriAbsL: scalable verification by abstraction and strategy prediction (competition contribution). In: Proc. TACAS (2). LNCS, vol. 13994, pp. 588–593. Springer, Berlin (2023). https://doi.org/10.1007/978-3-031-30820-8_41
66. Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schüssele, F.: Ultimate Taipan with symbolic interpretation and fluid abstractions (competition contribution). In: Proc. TACAS (2). LNCS, vol. 12079, pp. 418–422. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-45237-7_32
67. Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamee, D., Tomb, A.: Constructing semantic models of programs with the software analysis workbench. In: Proc. VSTTE. LNCS, vol. 9971, pp. 56–72. Springer, Berlin (2016). https://doi.org/10.1007/978-3-319-48869-1_5
68. D'Silva, V., Kröning, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **27**(7), 1165–1178 (2008). https://doi.org/10.1109/TCAD.2008.923410
69. Ermis, E., Hoenicke, J., Podelski, A.: Splitting via interpolants. In: Proc. VMCAI. LNCS, vol. 7148, pp. 186–201. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-27940-9_13
70. Ernst, G.: A complete approach to loop verification with invariants and summaries. Tech. Rep. (2020) https://doi.org/10.48550/arXiv.2010.05812. arXiv:2010.05812v2
71. Gadelha, M.Y., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via k-induction. Int. J. Softw. Tools Technol. Transf. **19**(1), 97–114 (2017). https://doi.org/10.1007/s10009-015-0407-9
72. 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). LNCS, vol. 11429, pp. 209–213. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-17502-3_15
73. Garavel, H., ter Beek, M.H., van de Pol, J.: The 2020 expert survey on formal methods. In: Proc. FMICS. LNCS, vol. 12327, pp. 3–69. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-58298-2_1
74. GDart-LLVM. https://github.com/tudo-aqua/gdart-llvm. Accessed: 2024–10–31
75. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proc. NDSS (2008). The Internet Society. https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/
76. Graves-Parallel. https://github.com/mgerrard/graves-par. Accessed: 2024–10–31
77. Greitschus, M., Dietsch, D., Podelski, A.: Loop invariants from counterexamples. In: Proc. SAS. LNCS, vol. 10422, pp. 128–147. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-66706-5_7
78. Gupta, A., Kahlon, V., Qadeer, S., Touili, T.: Model checking concurrent programs. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 573–611. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-10575-8_18
79. Hajdu, Á., Micskei, Z.: Efficient strategies for CEGAR-based model checking. J. Autom. Reason. **64**(6), 1051–1091 (2020). https://doi.org/10.1007/s10817-019-09535-x
80. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. LNCS, vol. 8044, pp. 36–52. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-39799-8_2
81. Heizmann, M., Chen, Y.F., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate automizer and the search for perfect interpolants (competition contribution). In: Proc. TACAS (2). LNCS, vol. 10806, pp. 447–451. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-89963-3_30

82. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. In: Proc. VMCAI. LNCS, vol. 5403, pp. 151–166. Springer, Berlin (2009). https://doi.org/10.1007/978-3-540-93900-9_15
83. Jaffar, J., Murali, V., Navas, J.A., Santosa, A.E.: TRACER: a symbolic execution tool for verification. In: Proc. CAV. LNCS, vol. 7358, pp. 758–766. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-31424-7_61
84. Jaffar, J., Maghareh, R., Godboley, S., Ha, X.L.: TracerX: dynamic symbolic execution with interpolation (competition contribution). In: Proc. FASE. LNCS, vol. 12076, pp. 530–534. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-45234-6_28
85. Jakobs, M.C., Richter, C.: CoVeriTest with adaptive time scheduling (competition contribution). In: Proc. FASE. LNCS, vol. 12649, pp. 358–362. Springer, Berlin (2021). https://doi.org/10.1007/978-3-030-71500-7_18
86. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4), 21 (2009). https://doi.org/10.1145/1592434.1592438
87. Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: Proc. VSTTE. LNCS, vol. 12031, pp. 1–18. Springer, Berlin (2019)
88. Kettl, M., Lemberger, T.: The static analyzer Infer in SV-COMP (competition contribution). In: Proc. TACAS (2). LNCS, vol. 13244, pp. 451–456. Springer, Berlin (2022). https://doi.org/10.1007/978-3-030-99527-0_30
89. Kröning, D., Tautschnig, M.: Cbmc: C bounded model checker (competition contribution). In: Proc. TACAS. LNCS, vol. 8413, pp. 389–391. Springer, Berlin (2014). https://doi.org/10.1007/978-3-642-54862-8_26
90. Lauko, H., Ročkai, P., Barnat, J.: Symbolic computation via program transformation. In: Proc. ICTAC. LNCS, vol. 11187, pp. 313–332. Springer, Berlin (2018). https://doi.org/10.1007/978-3-030-02508-3_17
91. Leeson, W., Dwyer, M.: Graves-CPA: a graph-attention verifier selector (competition contribution). In: Proc. TACAS (2). LNCS, vol. 13244, pp. 440–445. Springer, Berlin (2022). https://doi.org/10.1007/978-3-030-99527-0_28
92. Legion/SymCC. https://github.com/gernst/legion-symcc. Accessed: 2024–10–31
93. Lemberger, T.: Plain random test generation with PRTest (competition contribution). Int. J. Softw. Tools Technol. Transf. **23**(6), 871–873 (2021). https://doi.org/10.1007/s10009-020-00568-x
94. Liu, D., Ernst, G., Murray, T., Rubinstein, B.: Legion: best-first concolic testing (competition contribution). In: Proc. FASE. LNCS, vol. 12076, pp. 545–549. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-45234-6_31
95. Liu, D., Ernst, G., Murray, T., Rubinstein, B.I.P.: Legion: best-first concolic testing. In: Proc. ASE, pp. 54–65. IEEE (2020). https://doi.org/10.1145/3324884.3416629
96. Malík, V., Schrammel, P., Vojnar, T.: 2ls: heap analysis and memory safety (competition contribution). In: Proc. TACAS (2). LNCS, vol. 12079, pp. 368–372. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-45237-7_22
97. Manès, V.J.M., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: a survey. IEEE Trans. Softw. Eng. **47**(11), 2312–2331 (2021). https://doi.org/10.1109/TSE.2019.2946563
98. McMinn, P.: Search-based software test-data generation: a survey. Softw. Test. Verif. Reliab. **14**(2), 105–156 (2004). https://doi.org/10.1002/stvr.294
99. Metta, R., Medicherla, R.K., Chakraborty, S.: BMC+Fuzz: efficient and effective test generation. In: Proc. DATE, pp. 1419–1424. IEEE (2022). https://doi.org/10.23919/DATE54114.2022.9774672
100. Metta, R., Medicherla, R.K., Karmarkar, H.: VeriFuzz: fuzz centric test generation tool (competition contribution). In: Proc. FASE. LNCS, vol. 13241, pp. 341–346. Springer, Berlin (2022). https://doi.org/10.1007/978-3-030-99429-7_20
101. Monat, R., Ouadjaout, A., Miné, A.: Mopsa-C: modular domains and relational abstract interpretation for C programs (competition contribution). In: Proc. TACAS (2). LNCS, vol. 13994, pp. 565–570. Springer, Berlin (2023). https://doi.org/10.1007/978-3-031-30820-8_37
102. Nutz, A., Dietsch, D., Mohamed, M.M., Podelski, A.: Ultimate Kojak with memory safety checks (competition contribution). In: Proc. TACAS. LNCS, vol. 9035, pp. 458–460. Springer, Berlin (2015). https://doi.org/10.1007/978-3-662-46681-0_44
103. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y.L., Harman, M.: Chapter six - mutation testing advances: an analysis and survey. Adv. Comput. **112**, 275–378 (2019). https://doi.org/10.1016/bs.adcom.2018.03.015
104. Pasareanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. Int. J. Softw. Tools Technol. Transf. **11**(4), 339–353 (2009). https://doi.org/10.1007/s10009-009-0118-1
105. Richter, C., Wehrheim, H.: PeSCo: predicting sequential combinations of verifiers (competition contribution). In: Proc. TACAS (3). LNCS, vol. 11429, pp. 229–233. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-17502-3_19
106. Richter, C., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). https://doi.org/10.1007/s10515-020-00270-x
107. Ruland, S., Lochau, M., Jakobs, M.C.: HybridTiger: hybrid model checking and domination-based partitioning for efficient multi-goal test-suite generation (competition contribution). In: Proc. FASE. LNCS, vol. 12076, pp. 520–524. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-45234-6_26
108. Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.: Goblint: thread-modular abstract interpretation using side-effecting constraints (competition contribution). In: Proc. TACAS (2). LNCS, vol. 12652, pp. 438–442. Springer, Berlin (2021). https://doi.org/10.1007/978-3-030-72013-1_28
109. Scott, R., Dockins, R., Ravitch, T., Tomb, A.: Crux: Symbolic execution meets SMT-based verification (competition contribution). Zenodo (2022). https://doi.org/10.5281/zenodo.6147218
110. Test-comp 2023 benchmarks test tasks. https://test-comp.sosy-lab.org/2023/benchmarks.php. Accessed: 2024–10–31
111. Tóth, T., Hajdu, A., Vörös, A., Micskei, Z., Majzik, I.: Theta: a framework for abstraction refinement-based model checking. In: Proc. FMCAD, pp. 176–179 (2017). https://doi.org/10.23919/FMCAD.2017.8102257
112. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test-input generation with Java PathFinder. In: Proc. ISSTA, pp. 97–107. ACM, New York (2004). https://doi.org/10.1145/1007512.1007526
113. Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: the Goblint approach. In: Proc. ASE, pp. 391–402. ACM, New York (2016). https://doi.org/10.1145/2970276.2970337
114. Volkov, A.R., Mandrykin, M.U.: Predicate abstractions memory modeling method with separation into disjoint regions. Proc. Inst. Syst. Program. **29**, 203–216 (2017). https://doi.org/10.15514/ISPRAS-2017-29(4)-13
115. wasp. https://github.com/wasp-platform/wasp. Accessed: 2024–10–31