International Journal on Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

First International Competition on Software Testing

Dirk Beyer

Received: date / Revised version: date

Abstract Tool competitions are a special form of comparative evaluation, where each tool has a team of developers or supporters associated that makes sure the tool is properly configured to show its best possible performance. In several research areas, tool competitions have been a driving force for the development of mature tools that represent the state of the art in their field. This paper describes and reports the results of the 1st International Competition on Software Testing (Test-Comp 2019), a comparative evaluation of automatic tools for software test generation. Test-Comp 2019 was presented as part of TOOLympics 2019, a satellite event of the conference TACAS. Nine test generators were evaluated on 2 356 test-generation tasks. There were two test specifications, one for generating a test that covers a particular function call and one for generating a test suite that tries to cover the branches of the program.

Keywords Software testing · Test generation · Fuzzing · Program analysis · Bounded model checking · Test-suite validation · Competition · Test-Comp

1 Introduction

Software testing is as old as software development itself, because the easiest way to find out if software works is to test it. In the last few decades the tremendous breakthrough of theorem provers and satisfiability-modulo-theory (SMT) solvers have led to the development of efficient tools for automatic test generation. For example, symbolic execution and the idea to use it for test generation [30] exists for more than 40 years, but efficient implementations (e.g., Klee [16, 17]) had to wait for the availability of mature constraint solvers. Also, with the advent of automatic software model checking the opportunity to extract tests from counterexamples arose (see Blast [10] and JPF [36]). In the following years, many techniques from the areas of model checking and program analysis were adopted for the purpose of test generation and several strong hybrid combinations have been developed [23].

While several powerful software test generators are available [23], they are very difficult to compare. For example, a recent study [12] first had to develop a framework that supports to run test generators on the same program source code and to deliver tests in a common format for validation. Furthermore, there is no widely distributed benchmark suite available and neither input programs nor output test suites follow a standard format. In software verification, the competition SV-COMP [5] helped to overcome similar problems: the competition community developed standards for defining nondeterministic functions and a language to write specifications (so far for C and Java programs) and established a standard exchange format for the output (witnesses). The competition also helped to adequately give credits to PhD students and PostDocs for their engineering efforts and technical contributions. A competition event with high visibility can foster the transfer of theoretical and conceptual advancements in software testing into practical tools, and also gives credits and benefits to students who spend considerable amounts of time developing testing algorithms and software tools. Successful participation in competitions indicates qualification.

This work was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 418257054 (Coop).

Open Access was funded by Projekt DEAL.

A preliminary version was published in Proc. TACAS 2019 [6].

D. Beyer
LMU Munich, Oettingenstr. 67, 80538 Munich, Germany

Comparative overviews are helpful for engineers when selecting test tools for their purpose.

Test-Comp is designed to compare automatic state-of-the-art software test generators with respect to effectiveness and efficiency. This comprises a preparation phase in which a set of benchmark programs is collected and classified (according to application domain, kind of bug to find, coverage criterion to fulfill, theories needed), in order to derive competition categories. After the preparation phase, the tools are submitted, installed, and run on the set of benchmark tasks.

Test-Comp uses the benchmarking framework BenchExec [14], which is already successfully used in other competitions, most prominently, all competitions that run on the StarExec infrastructure [35]. Similar to SV-COMP, the test generators in Test-Comp are applied to programs in a fully automatic way. The results are collected via the BenchExec results format, and transformed into tables and plots in several formats.

Competition Goals. In summary, the most important goals of the competition Test-Comp are the following:

- Establish a set of benchmarks for software testing in the community. This means to create and maintain a set of well-defined programs together with coverage criteria, and to make those publicly available for researchers to be used in performance comparisons when evaluating a new algorithm, technology, or implementation.
- Establish standards for software test generation. This means, most prominently, to develop a standard for marking input values in programs, define an exchange format for test suites, and agree on a specification language for test-coverage criteria. Furthermore, we define how to validate the resulting test suites.
- Provide an overview of available tool implementations for test generation and a snapshot of the state-of-the-art in software-testing research to the community. This means to compare, independently from particular paper projects and specific techniques, different test generators in terms of effectiveness and performance on a large benchmark set.
- Increase the visibility and credits that tool developers receive. This means to provide a forum for presentation of tools and discussion of the latest technologies, and to give the developers (often PhD students) the opportunity to publish about the development work that they have done.
- Educate PhD students and other participants on how to set up performance experiments, packaging tools in a way that supports reproducibility, and how to perform a robust and accurate research experiment.
- Provide resources to development teams that do not have sufficient computing resources available and give them the opportunity to obtain performance results from experiments on large benchmark sets.
- Establish a transparent process to enable the test-generation community to be the driving force behind the competition.

Related Competitions. In other areas, there are several established competitions. For example, there are three competitions in the area of software verification: (i) a competition on automatic verifiers under controlled resources (SV-COMP [5]), (ii) a competition on verifiers with arbitrary environments (RERS [26]), and (iii) a competition on (interactive) verification (VerifyThis [27]). In software testing, there are several competition-like events, for example, the DARPA Cyber Grand Challenge [34] [1], the IEEE International Contest on Software Testing [2], the Software Testing World Cup [3], and the Israel Software Testing World Cup [4]. Those contests are organized as on-site events, where teams of people interact with certain testing platforms in order to achieve a certain coverage of the software under test.

There are two competitions for automatic and off-site testing: Rode0day [5] is a competition that is meant as a continuously running evaluation on bug-finding in binaries (currently Grep and SQLite). The unit-testing tool competition [29] [6] is part of the SBST workshop and compares tools for unit-test generation on Java programs.

So far, there was no comparative evaluation of automatic test generators in a controlled environment in which the tool developers were involved as participants and jury. Test-Comp [6] [7] is meant to close this gap. The results of the first edition of Test-Comp were presented as part of the TOOLympics 2019 event [1], where 16 competitions in the area of formal methods were presented.

2 Organizational Classification and Schedule

The competition Test-Comp is designed according to the model of SV-COMP [2], the International Competition on Software Verification.

Classification. Test-Comp shares the following organizational principles:

- **Automatic:** The tools are executed in a fully automated environment, without any user interaction.

[1] https://www.darpa.mil/program/cyber-grand-challenge/
[2] http://paris.utdallas.edu/qrs18/contest.html
[3] http://www.softwaretestingworldcup.com/
[4] https://www.inflectra.com/Company/Article/480.aspx
[5] https://rode0day.mit.edu/
[6] https://sbst19.github.io/tools/
[7] https://test-comp.sosy-lab.org/

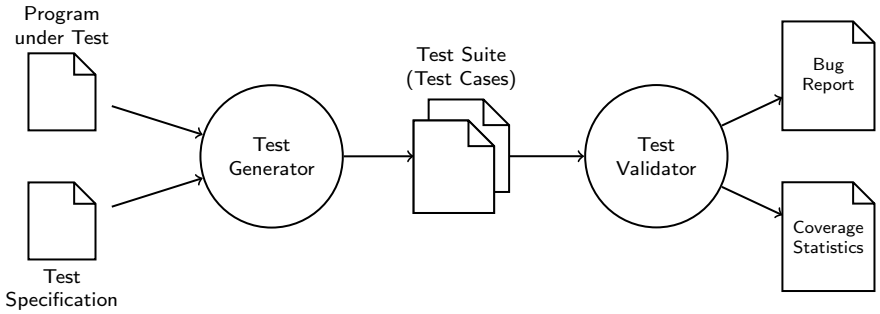

Fig. 1: Flow of the Test-Comp execution for one test generator; the left side depicts a test-generation run; the right side depicts a test-validation run

- **Off-site:** The competition takes place independently from a conference location, in order to flexibly allow problem solving and organizational changes.
- **Reproducible:** The experiments are controlled and reproducible, that is, the resources are limited, controlled, measured, and logged.
- **Jury:** The jury is the advisory board of the competition, is responsible for qualification decisions on tools and benchmarks, and serves as program committee for the reviewing and selection of papers to be published in conference proceedings or a journal. The jury ensures transparency of the competition organization and judges qualification of participants (but not their performance, which is computed using a scoring schema from the results, see Sect. 5). The jury is also responsible for new competition rules and deciding on new categories.
- **Training:** The competition flow includes a training phase during which the participants get a chance to train their tools on the potential benchmark instances and during which the organizer ensures a smooth competition execution, giving preliminary feedback to the participating teams.

Schedule. A typical Test-Comp schedule has the following deadlines and phases:

- **Call for Participation:** The organizer announces the competition on the mailing list.[8]
- **Registration of Participation and Training Phase:** The tool developers register for participation and submit a first version of their tool together with documentation to the competition. The tool can later be updated and is used for pre-runs by the organizer and for qualification assessment by the jury. Preliminary results are reported to the tool developers, and made available to the jury.
- **Final-Version Submission and Evaluation Phase:** The tool developers submit the final versions of their tools. The benchmarks are executed using the submitted tools and the experimental results are reported to the authors. Final results are reported to the tool developers for inspection and are made publicly available only after team approval.
- **Results Announced:** The organizer announces the results on the competition web site.
- **Publication:** The competition organizer writes the competition report, the tool developers write the tool description and participation reports. The jury reviews the papers and the competition report.

3 Rules and Definitions

Test-Generation Task. A *test-generation task* is a pair of an input program (program under test) and a test specification. A *test-generation run* is a non-interactive execution of a test generator on a single test-generation task, in order to generate a test suite according to the test specification. A *test suite* is a sequence of tests, given as a directory of files according to the format for exchangeable test suites.[9] A *test-validation run* is a non-interactive execution of a test validator on a given test suite, in order to evaluate a test suite according to the test specification.

Execution of a Test Generator. Figure 1 illustrates the process of executing one test generator on one test-generation task. One test-generation run for a test generator gets as input (i) a program from the benchmark suite and (ii) a test specification (find bug, or coverage criterion), and returns as output a test suite (i.e., a set of tests). The test generator is contributed by the competition participant. The test-generation runs are

[8] https://groups.google.com/forum/#!forum/test-comp

[9] https://gitlab.com/sosy-lab/software/test-format/

Table 1: Coverage specifications used in Test-Comp 2019

Category	Test Specification	Interpretation
Cover-Error	`COVER( init(main()), FQL(COVER EDGES(@CALL(__VERIFIER_error))) )`	The test suite contains at least one test that executes function `__VERIFIER_error`.
Cover-Branches	`COVER( init(main()), FQL(COVER EDGES(@DECISIONEDGE)) )`	The test suite contains tests such that all branches of the program are executed.

executed centrally by the competition organizer. The test validator takes as input the test suite from the test generator and validates it by executing the program on all test-generation tasks: for bug finding it checks if the bug is exposed and for coverage it reports the coverage using the GNU tool `gcov`.[10]

Test Specification. Table 1 lists the two test specifications that are used in Test-Comp 2019 and constitute the two main competition categories. The first describes a formula that is typically used for bug finding: the test generator should find a test that executes a certain error function (*Cover-Error*). The second describes a formula that is used to obtain a standard test suite for quality assurance: the test generator should find a test suite for branch coverage (*Cover-Branches*). The specification for testing a program is given to the test generator as input file (either `properties/coverage-error-call.prp` or `properties/coverage-branches.prp` for Test-Comp).

The definition `init(main())` is used to define the entry of the program under test. The definition `FQL(f)` specifies that coverage definition `f` should be achieved. The FQL (FShell query language [25]) coverage definition `COVER EDGES(@DECISIONEDGE)` means that all branches should be covered, `COVER EDGES(@BASICBLOCKENTRY)` means that all statements should be covered, and `COVER EDGES(@CALL(__VERIFIER_error))` means that function `__VERIFIER_error` should be called. A complete specification looks like those in Table 1.

License Requirements for Submitted Test-Generator Archives. The test generators need to be publicly available for download as binary archive under a license that allows the following (cf. [5]):

- reproduction and evaluation by anybody (including results publication),
- no restriction on the usage of the verifier output (log files, witnesses), and
- any kind of (re-)distribution of the unmodified verifier archive.

Qualification. Before a tool or person can participate in the competition, the jury evaluates the following qualification criteria.

Tool. A test tool is qualified to participate as competition candidate if the tool is (a) publicly available for download and fulfills the above license requirements, (b) works on the GNU/Linux platform (more specifically, it must run on an x86_64 machine), (c) is installable with user privileges (no root access required, except for required standard Ubuntu packages) and without hard-coded absolute paths for access to libraries and non-standard external tools, (d) succeeds for more than 50 % of all training programs to parse the input and start the test process (a tool crash during the test-generation phase does not disqualify), and (e) produces test suites that adhere to the exchange format (see above).

Person. A person (participant) is qualified as competition contributor for a competition candidate if the person (a) is a contributing designer/developer of the submitted competition candidate (witnessed by occurrence of the person's name on the tool's project web page, a tool paper, or in the revision logs) or (b) is authorized by the competition organizer (after the designer/developer was contacted about the participation).

4 Benchmark Programs and Categories

The first edition of Test-Comp is based on programs written in the programming language C. The input programs are taken from the largest and most diverse open-source repository of software verification and test-generation tasks [11], which is also used by SV-COMP [5].

Selection. We selected all programs for which the following properties were satisfied (cf. issue on GitHub [12]):

1. compiles with `gcc`, if a harness for the special input-providing nondeterministic methods is provided,
2. contains at least one call to a such an input-providing nondeterministic function,
3. does not rely on nondeterministic pointers,
4. does not have expected verdict `false` for a 'termination' specification, and
5. has expected verdict `false` for an 'unreach-call' specification (only for category *Cover-Error*).

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

[11] https://github.com/sosy-lab/sv-benchmarks

[12] https://github.com/sosy-lab/sv-benchmarks/pull/774

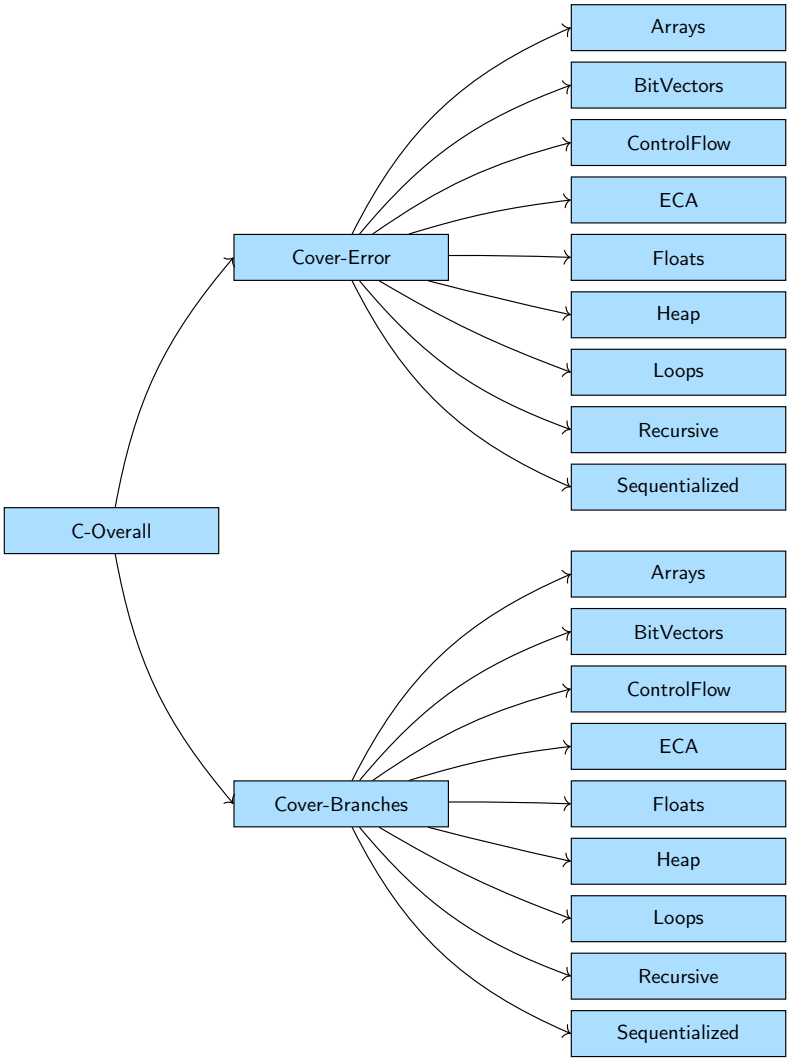

Fig. 2: Category structure for Test-Comp 2019

This selection yields a total of 2 356 test-generation tasks, namely 636 test-generation tasks for category *Cover-Error* and 1 720 test-generation tasks for category *Cover-Branches*.

We now explain the above requirements in more detail:

(1) It is necessary to be able to compile and link the program because we can execute a program only if all declared functions are implemented. (We need to execute the compiled programs on tests in order to measure coverage to evaluate the test suites produced by the test generators.) According to the specification of the benchmark repository, there are several unimplemented functions, which are meant to feed test inputs. Those functions have a name of the form `__VERIFIER_nondet_X()`, where `X` is a type from the set {`bool`, `char`, `int`, `float`, `double`, `loff_t`, `long`, `pchar`, `pthread_t`, `sector_t`, `short`, `size_t`, `u32`, `uchar`, `uint`, `ulong`, `unsigned`, and `ushort`} and the implementation can be assumed to return an arbitrary (nondeterministic) value of that type, without any side effects.

(2) The programs that we use for Test-Comp need to have at least one call of such a function that returns a nondeterministic value, in order to be able to identify the test inputs of the program and to later feed the test values to the program when executing it.

(3) The specification of the benchmark repository also knows special functions to return nondeterministic values for pointers (type `void *`), which are meant for verification based on model checking (used in SV-COMP). We do not use programs with such function calls in Test-Comp, because they often introduce undefined behavior. Those calls will be eliminated in the benchmark repository in the future, from 2020 onwards, in order to avoid undefined behavior in verification and test-generation tasks.

(4) We exclude from Test-Comp all programs in the benchmark repository that have non-terminating executions. Those programs are meant for evaluating verification tools that detect nontermination. The verdict for the behavioral specification for termination is available in the task-definition files in the repository.
(5) For category *Cover-Error*, the task definition needs to contain the verdict `false` for the behavioral specification that a certain function call is not reachable. Otherwise, if the call is not reachable, then the task is not relevant for category *Cover-Error* of Test-Comp.

Categories. The test-generation tasks are partitioned into categories. Figure 2 illustrates the structure of the category composition. The results (in Tables 6 and 7) are listed according to the main categories. Category *C-Overall* consists of the two main categories *Cover-Error* and *Cover-Branches* (according to Table 1), which in turn consist of the following subcategories (same for both main categories in 2019): *Arrays, BitVectors, ControlFlow, ECA, Floats, Heap, Loops, Recursive,* and *Sequentialized*. The detailed definition of the categories (which test-generation tasks are contained in which subcategory) is available on the competition web site.[13]

The main categories partition the test-generation tasks according to the test specification, that is, whether to generate a test suite for covering a single bug or to generate a test suite for covering as many branches as possible. The subcategories are structured based on the features of the programs that the test generators need to support: programs with arrays, with bit-vector arithmetic that cannot be approximated as linear arithmetic, with control-flow that matters for the behavior, with a certain style of programming for event-condition-action (ECA) systems, with floating-point arithmetics, with data structures on the heap, with loops that are important to be analyzed, with recursive function calls, and programs that result from a transformation of multi-threaded programs to sequential programs.

The benchmark collection SV-Benchmarks contains benchmark sets of C programs (`c/`), Java programs (`java/`), and Horn clauses (`clauses/`). Test-Comp 2019 used only programs written in C. The C collection consists of many subdirectories, in order to structure the programs according to their provenance and features. Each directory usually contains a `README` file with a description of the contents and a `LICENSE` file (link) to declare the license of the programs. The subcategories are defined in category-definition files (`.set`). For example, the subcategory *Arrays* is defined by the file `c/ReachSafety-Arrays.set`. The above-mentioned web page[13] is generated from those category-definition files. The category-configuration files (`.cfg`) provide a short description of the subcategory and important information about the programs in the subcategory, most importantly, the bit architecture. For example, the category configuration for subcategory *Arrays* is contained in the file `c/ReachSafety-Arrays.cfg`.

[13] https://test-comp.sosy-lab.org/2019/benchmarks.php

5 Scoring Schema

Every test-generation run will be executed in the execution environment of the competition according to the flow in Fig. 1, which produces for every test generator and every test-generation task (which is a pair of a C program and a test specification) a coverage value, which is a value in the interval [0, 1]. The coverage values are also called score points.

Evaluation by scores and runtime. The participating test generators are ranked according to the cumulative coverage (sum of score points). Test generators with the same cumulative coverage are ranked according to success-runtime. The success-runtime for a test generator is the total CPU time over all test-generation runs for which the test generator successfully produced a test suite.

Cover-Error. The first category is to show the abilities to discover bugs. The benchmark set consists of programs that contain a bug. The coverage value is defined to be either 0 or 1, as follows:

1	if the program under test is executed on a generated test that explores the bug (i.e., specified function was called)
0	otherwise

Cover-Branches. The second category is to cover as many branches as possible. The coverage criterion was chosen because many test generators support this standard criterion by default. Other coverage criteria can be reduced to branch coverage by transformation [24]. The coverage value (as reported by `gcov`[10]; value from [0, 1]) represents the ratio of branches of the program that are covered by the generated tests to the number of all branches of the program. The coverage value is defined as follows:

c	if the program under test is executed on all generated tests and c is the coverage value as measured with the tool `gcov`

Note that we measured what `gcov` calls branch coverage. In our experiments we discovered that what `gcov` reports is in fact not branch coverage, but measurement values that are closer to what is usually referred to as condition coverage. Therefore, the next Test-Comp

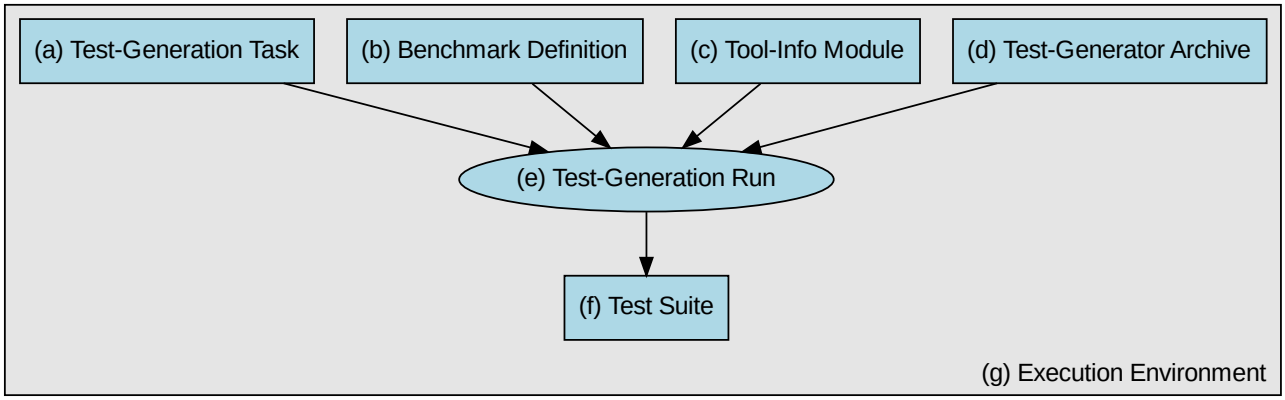

Fig. 3: Test-Comp components and the execution flow; in relation to Fig. 1, the program under test and test specification are defined by the test-generation task (a), the test generator is taken from the test-generator archive (d), and the test suite is stored in an archive (f) for later evaluation by a test-validation run, which also works with the components depicted in the above figure

uses measurements as reported by TestCov [13], which implements the usual definition of branch coverage.

Opt Out. It is possible for participants to opt out from certain categories that are not supported by the test generator. In this case, the tables would show no result (empty table cell). In Test-Comp 2019, all teams participated in all categories. Esbmc did not support branch coverage and therefore the table displays a zero as result for category *Cover-Branches* (see Table 6).

Normalization of Scores. Since the main categories are composed of subcategories, and the subcategories contain different numbers of test-generation tasks, there would be a bias towards subcategories with a large number of test-generation tasks. In other words, without normalization, it would maximize the score to work on categories that consist of many similar programs. However, we do not want to stipulate that one category is more important than another. Thus, we need to normalize the score, such that all subcategories have the same influence on the final result. The goal is to reduce the influence of a test-generation task in a large category compared to a test-generation task in a small category, and thus, balance over the categories. We use the normalization that is also used by SV-COMP (see competition report of SV-COMP 2013 [3], page 597): The *score for a meta category* is computed from the scores of all k contained (sub-) categories using a normalization by the number of contained test-generation tasks: The normalized score sn_i of a test generator in category i is obtained by dividing the score s_i by the number of tasks n_i in category i ($sn_i = s_i/n_i$), then the sum $\Sigma_{i=1}^{k} sn_i$ over the normalized scores of the categories is multiplied by the average number of tasks per category. An example calculation can be found on the web page of SV-COMP.[14]

6 Components for Reproducibility

Reproducibility of the results is a main concern of a competition like Test-Comp. The competition must be as transparent and reproducible as possible. To achieve this goal, we duplicate the setup from SV-COMP [4] and describe here our adaptation to Test-Comp. We have to try to control all variables that might influence the results.

Figure 3, in its top row, shows the input of the process of executing a test-generation run of the competition: (a) the test-generation task, (b) the benchmark definition, (c) the tool-info module, and (d) the test-generator archive. Using those four inputs, (e) the test-generation run produces (f) the resulting test suite in (g) the execution environment. Table 2 provides for each of the 7 components the repository URL and the tag to identify the precise version that was used in the competition. Table 3 lists the archives that were published on Zenodo.

Repository of Test-Generation Tasks (a). The repository of test-generation tasks [11] is maintained by the community, using the GitHub issue tracker and pull requests to efficiently handle contributions from the contributors. The repository has more than 80 contributors.[15] Continuous-integration ensures that the programs are

[14] https://sv-comp.sosy-lab.org/2019/rules.php#meta
[15] https://github.com/sosy-lab/sv-benchmarks/graphs/contributors

Table 2: Publicly available components for reproducing Test-Comp 2019

Component	Fig. 3	Repository	Version
Test-Generation Tasks	(a)	github.com/sosy-lab/sv-benchmarks	testcomp19
Benchmark Definitions	(b)	gitlab.com/sosy-lab/test-comp/bench-defs	testcomp19
Tool-Info Modules	(c)	github.com/sosy-lab/benchexec	1.18
Test-Generator Archives Test-Validator Archive	(d)	gitlab.com/sosy-lab/test-comp/archives-2019	testcomp19
Benchmarking	(e)	github.com/sosy-lab/benchexec	1.18
Test-Suite Format	(f)	gitlab.com/sosy-lab/test-comp/test-format	testcomp19
Execution Environment	(g)	gitlab.com/sosy-lab/test-comp/bench-defs	testcomp19

Table 3: Artifacts archived for Test-Comp 2019

Content	DOI	Ref.
Test-Generation Tasks	10.5281/zenodo.3856478	[8]
Competition Results	10.5281/zenodo.3856661	[7]
Test Suites (Witnesses)	10.5281/zenodo.3856669	[9]
BenchExec 1.18	10.5281/zenodo.2561835	[14, 37]

Table 4: Execution limits for each run in Test-Comp '19

Limit	Test-Suite Generation	Test-Suite Validation
CPU Time	15 min	3 h
RAM	15 GB	7 GB
Processing Units	8	2

compilable by GCC and CLANG. The test-generation tasks as used for Test-Comp 2019 are tagged in the repository and archived at Zenodo [8].

The repository describes test-generation tasks using a task-definition file in YAML format, according to the standard: https://gitlab.com/sosy-lab/benchmarking/task-definition-format. For example, the task-definition file c/ntdrivers-simplified/cdaudio_simpl1.cil-1.yml refers to the C program (extracted from a device driver) c/ntdrivers-simplified/cdaudio_simpl1.cil-1.c and several test specifications, including c/properties/coverage-branches.prp, which results in a test-generation task that consists of this C program and the test specification to generate a test suite that covers all branches of the program.

Benchmark Definitions (b). For executing test-generation runs, we need to set resource limits, and we need to know for each test generator, (i) which test-generation tasks need to be given to the test generator as input and (ii) which parameters need to be passed to the test generator (there are global, test-generator-specific parameters to be passed to the tool, and there is one task-specific parameter: the bit architecture). The benchmark definitions are XML files in the format that BENCHEXEC expects; they are available in a repository. The execution of each test-generation run was limited to the resources specified in Table 4, for CPU time, RAM, and number of processing units (cores) of the CPU.

For example, the benchmark definition for COVERITEST is shown in Fig. 4 (also available in the repository as benchmark-defs/coveritest.xml). This XML file describes first the tool-info module to be used (`tool="cpachecker"`, see below under (c)), followed by a display name and the resource limits from Table 4. It also specifies the CPU model (`cpuModel="Intel Xeon E3-1230 v5 @ 3.40 GHz"`) and that all 8 CPU cores shall be reserved for the test-generation run (`cpuCores="8"`). The rest of the file specifies the result files, the options for COVERITEST, the properties, and the programs (compare with Fig. 2). A more detailed description is available in the BENCHEXEC repository (doc/benchexec.md#defining-tasks-for-benchexec).

Tool-Specific Information (c). In order to correctly execute a test generator, we need to provide a tool-info module to BENCHEXEC. The tool-info module assembles the command-line to properly invoke the test generator (including program-source and test-specification files as well as the parameters) from the parts specified in the benchmark definition (b). The tool-info modules that were used in Test-Comp 2019 are available in BENCHEXEC release 1.18 [37].

Test-Generator Archives / Test-Validation Archive (d). The test generators are provided in an archive containing a license (that permits distribution, use in Test-Comp, and reproducing the results) and all parts that are needed to execute the test generator (statically-linked executables, all components for which a certain version is required, or for which no standard Ubuntu package is available, are included). The test generators and the above-mentioned components are provided in the Test-Comp archives repository. The same holds for the test-validator.

```
<?xml version="1.0"?>
<!DOCTYPE benchmark PUBLIC "+//IDN sosy-lab.org//DTD BenchExec benchmark 1.9//EN" "http://www.sosy-lab.org/benchexec/benchmark-1.9.dtd">
<benchmark tool="cpachecker" displayName="CPA/CoVeriTest" timelimit="900 s" memlimit="15 GB" cpuCores="8">

  <require cpuModel="Intel Xeon E3-1230 v5 @ 3.40 GHz" cpuCores="8"/>

  <resultfiles>**test-suite/*</resultfiles>

  <option name="-benchmark"/>
  <option name="-heap">10000M</option>
  <option name="-testcomp19"/>

  <rundefinition name="test-comp19_prop-coverage-error-call">
    <propertyfile>../sv-benchmarks/c/properties/coverage-error-call.prp</propertyfile>
  </rundefinition>

  <rundefinition name="test-comp19_prop-coverage-branches">
    <propertyfile>../sv-benchmarks/c/properties/coverage-branches.prp</propertyfile>
  </rundefinition>


  <tasks name="ReachSafety-Arrays">
    <includesfile>../sv-benchmarks/c/ReachSafety-Arrays.set</includesfile>
    <option name="-32"/>
  </tasks>
  <tasks name="ReachSafety-BitVectors">
    <includesfile>../sv-benchmarks/c/ReachSafety-BitVectors.set</includesfile>
    <option name="-32"/>
  </tasks>
  <tasks name="ReachSafety-ControlFlow">
    <includesfile>../sv-benchmarks/c/ReachSafety-ControlFlow.set</includesfile>
    <option name="-32"/>
  </tasks>
  <tasks name="ReachSafety-ECA">
    <includesfile>../sv-benchmarks/c/ReachSafety-ECA.set</includesfile>
    <option name="-32"/>
  </tasks>
  <tasks name="ReachSafety-Floats">
    <includesfile>../sv-benchmarks/c/ReachSafety-Floats.set</includesfile>
    <option name="-32"/>
  </tasks>
  <tasks name="ReachSafety-Heap">
    <includesfile>../sv-benchmarks/c/ReachSafety-Heap.set</includesfile>
    <option name="-32"/>
  </tasks>
  <tasks name="ReachSafety-Loops">
    <includesfile>../sv-benchmarks/c/ReachSafety-Loops.set</includesfile>
    <option name="-32"/>
  </tasks>
  <tasks name="ReachSafety-Recursive">
    <includesfile>../sv-benchmarks/c/ReachSafety-Recursive.set</includesfile>
    <option name="-32"/>
  </tasks>
  <tasks name="ReachSafety-Sequentialized">
    <includesfile>../sv-benchmarks/c/ReachSafety-Sequentialized.set</includesfile>
    <option name="-32"/>
  </tasks>

</benchmark>
```

Fig. 4: Benchmark definition benchmark-defs/coveritest.xml for test generator CoVeriTest

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE test-metadata SYSTEM "https://gitlab.com/sosy-lab/software/test-format/blob/master/test-metadata.dtd">
<test-metadata>
        <sourcecodelang>C</sourcecodelang>
        <producer>CPAchecker 1.8-svn 30375</producer>
        <specification>COVER( init(main()), FQL(COVER EDGES(@DECISIONEDGE)) )</specification>
        <programfile>../../sv-benchmarks/c/ntdrivers-simplified/cdaudio_simpl1.cil-1.c</programfile>
        <programhash>eff1925cc737e819caaeac8669d2e9012af323aabffe91551e0a9b218d320618</programhash>
        <entryfunction>main</entryfunction>
        <architecture>32bit</architecture>
        <creationtime>2019-02-06T01:13:08+01:00</creationtime>
</test-metadata>
```

Fig. 5: Meta data of a test suite from Test-Comp 2019 (taken from test suite 1bbef0df...zip)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE testcase SYSTEM "https://gitlab.com/sosy-lab/software/test-format/blob/master/testcase.dtd">
<testcase>
        <input>8</input>
        <input>0</input>
        <input>9</input>
        <input>10</input>
        <input>11</input>
        <input>12</input>
</testcase>
```

Fig. 6: Test of a test suite from Test-Comp 2019 (taken from test suite 1bbef0df...zip)

Precise Controlling and Measurement of Resources (e). For scientifically valid experiments, we require for each test-generation run a reliable assignment and controlling of computing resources (cores, memory, CPU time), and a precise measurement. There are several requirements that experiments of a competition such as Test-Comp have to fulfill [14]: (i) accurate measurement and reliable enforcement of limits for CPU time and memory, (ii) reliable termination of processes (including all child processes), (iii) correct assignment of local memory (for NUMA architectures), and (iv) isolation of the test-generation run in a container. We used BenchExec [14] to perform all Test-Comp experiments, because this benchmarking framework lets us conveniently benefit from the modern resource-control and measurement mechanisms that the Linux kernel offers. All results, including raw measurement results, log files, and HTML files, are archived at Zenodo [7].

Test Suites (f). The ranking of test generators in Test-Comp is based on achieved coverage for the given test-generation tasks. That is, given an input program and a test specification, the test generator has to produce a test suite that covers the test specification as much as possible. The test suite functions as witness for the achieved coverage, needs to be stored and evaluated. Test suites are stored in a community-agreed test-suite format (gitlab.com/sosy-lab/test-comp/test-format). All test suites that were produced in Test-Comp 2019 are archived at Zenodo [9].

For example, the test suite that CoVeriTest generated for the above-mentioned test-generation task is directly accessible also on the Test-Comp web site (visit https://test-comp.sosy-lab.org/2019/results/results-verified/, click on the score (14) for column CoVeriTest and row `coverage-branches.ReachSafety-ControlFlow`, then in the table with the detailed results click on the cell for column `test-suite` and row `ntdrivers-simplified/cdaudio_simpl1.cil-1.yml`, to obtain the file 1bbef0df...zip). The test suite is contained in a directory `test-suite/` inside the ZIP archive. The directory contains a file `metadata.xml` that describes the test suite and one file `test...xml` for each test (also called test vector). The meta-data file is shown in Fig. 5; it provides information about the language of the program, the producing engine (CoVeriTest is based on CPAchecker), the test specification, the program path, the SHA-256 hash of the program, the entry function, the data model of the CPU for which the program was written, and the creation time stamp. The first test of the test suite is shown in Fig. 6; it provides a sequence of test values to be fed into the program.

Table 5: Competition candidates with tool references and representing jury members

Participant	Ref.	Jury Member	Affiliation
CoVeriTest	[11, 28]	Marie-Christine Jakobs	LMU Munich, Germany
CPA/Tiger-MGP	[15, 33]	Sebastian Ruland	TU Darmstadt, Germany
Esbmc-bkind	[21, 22]	Rafael Menezes	Federal U. of Amazonas, Brazil
Esbmc-falsif	[21]	Mikhail Gadelha	University of Southampton, UK
FairFuzz	[32]	Caroline Lemieux	University of California at Berkeley, USA
Klee	[17]	Cristian Cadar	Imperial College London, UK
PRTest	[31]	Thomas Lemberger	LMU Munich, Germany
Symbiotic	[18, 19]	Martina Vitovská	Masaryk U., Czechia
VeriFuzz	[20]	Raveendra Kumar M.	Tata Consultancy Services, India

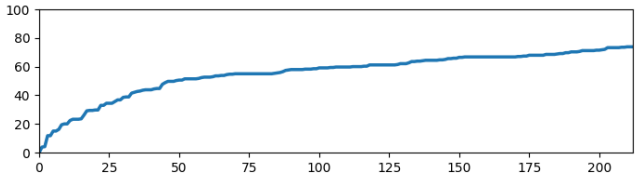

Fig. 7: Coverage plot for the discussed test suite from Test-Comp 2019 (test suite 1bbef0df...zip); the diagram shows the number of tests processed on the x-axis and the coverage in percent on the y-axis, that is, a data point (100, 60) informs us that the first 100 tests cover 60 % of the program's branches.

The discussed test suite contains 212 tests. During the test-validation run, the test validator takes the test suite and executes each test of the program (feeding in the values from the XML file). For the discussed test suite, the test generator is assigned a score of 0.738, because the test suite covers 73.8 % of all branches of the program. The increase in coverage by each test is illustrated in Fig. 7.

Execution Environment (g). The machines for running the experiments were part of a compute cluster at LMU Munich that consists of 168 machines; each test-generation run was executed on an otherwise completely unloaded, dedicated machine, in order to achieve precise measurements. Each machine had one Intel Xeon E3-1230 v5 CPU, with 8 processing units each, a frequency of 3.4 GHz, 33 GB of RAM, and a GNU/Linux operating system (x86_64-linux, Ubuntu 18.04 with Linux kernel 4.15). Further technical parameters of the competition machines are available in the file `README.md` of the repository that also contains the benchmark definitions. The job-distribution system VerifierCloud [16] was used to distribute, install, run, and clean-up test-generation runs, and to collect the results.

7 Results

For the first time, the competition Test-Comp 2019 presents the state of the art in fully automatic test-generation for whole C programs, using a developer-involved comparative evaluation based on controlled experiments. The results help in understanding the current achievements of the test-generation research, in terms of effectiveness (test coverage, as accumulated in the score) and efficiency (resource consumption in terms of CPU time). All results mentioned in this article were inspected and approved by the participants.

Participating Tools. The automatic test-generators that participated in the first edition of Test-Comp are listed in Table 5. The table provides for each of the 9 participating systems the test-generator name (links to the project web site in the PDF version of this article), references to system descriptions, and the representing jury member, with affiliation).

Quantitative Results. Table 6 presents the quantitative overview of all tools and all categories. The head row mentions the category and the number of test-generation tasks in that category. The tools are listed in alphabetical order; every table row lists the scores of one test generator. We indicate the top three candidates by formatting their scores in bold face and in larger font size. More information (including interactive tables, quantile plots for every category, and also the raw data in XML format) is available on the competition web site [17] and in the results artifacts (see Table 3). Table 7 reports the top three test generators for each category. The consumed run time (column 'CPU Time') is given in hours and the consumed energy (column 'Energy') is given in kW h.

Score-Based Quantile Functions. We use score-based quantile functions (see [14], Sect. 7.7 and 7.8, and [4], pages 899–900) for quality assessment, because these visualizations make it easier to understand the

[16] https://vcloud.sosy-lab.org

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

Table 6: Quantitative overview; main categories

Test Generator	Cover-Error 636 tasks	Cover-Branches 1720 tasks	C-Overall 2356 tasks
CoVeriTest	**397**	**1153**	**1524**
CPA/Tiger-MGP	361	966	1331
Esbmc-bkind	237	0	438
Esbmc-falsif	247	0	457
FairFuzz	365	874	1275
Klee	**499**	**1226**	**1764**
PRTest	193	476	683
Symbiotic	365	907	1298
VeriFuzz	**595**	**1238**	**1951**

Table 7: Overview of the top-three verifiers for each category (CPU time in h, with two significant digits)

Rank	Test Generator	Score	CPU Time (in h)	Energy (in kW h)
Cover-Error				
1	VeriFuzz	**595**	15	.18
2	Klee	499	4.9	.040
3	CoVeriTest	397	8.7	.080
Cover-Branches				
1	VeriFuzz	**1238**	430	5.5
2	Klee	1226	310	2.9
3	CoVeriTest	1153	340	3.9
Overall				
1	VeriFuzz	**1951**	440	5.7
2	Klee	1764	320	2.9
3	CoVeriTest	1524	350	4.0

results of the comparative evaluation. The web site [17] and the results artifact (Table 3) include such a plot for each category. As example, we show the plot for category *C-Overall* (all test-generation tasks) in Fig. 8. All 9 test generators participated in category *C-Overall*, for which the quantile plot shows the overall performance over all categories (scores for meta categories are normalized, see Sect. 5).

Generation of Score-Based Quantile Plots. For the score calculation, we have computed a function that maps each test-generation task to the coverage (error coverage or branch coverage) of the test suite that was generated for this test-generation task, and another function to map each test-generation task to the achieved normalized coverage score. Now, we sort the pairs ⟨test-generation task, achieved normalized score⟩ by the score in descending order, and accumulate the score and test-generation tasks. The pairs ⟨cumulative score, number test-generation tasks⟩ define the quantile function, which maps an achieved normalized score to the minimal number of test-generation tasks that are needed to achieve this coverage score with the given test suite. (Note that quantile plots compare quantiles and not individual test-generation tasks, that is, one cannot tell from a quantile plot the performance on a certain single test-generation task.) Plots of functions like this can easily be generated using tools like `gnuplot`.[18]

Interpretation of Data Points. A data point (x, y) for a test generator tells us that the tool needed y test-generation tasks to achieve the score of x score points (cumulative, normalized coverage values). For example, the quantile function for VeriFuzz contains the pair $(1802.9, 1000)$, which means that VeriFuzz needs at least 1000 test-generation tasks to achieve a coverage of 1802.9 score points. Such plots make it easy to compare the performance of different test generator, because the graphs are monotonically increasing. The lower and the more to the right a graph is drawn, the better is the test generator.

Overall Quality Measured in Scores (Right End of Graph). VeriFuzz is the winner of this category: the x-coordinate of the right-most data point represents the highest total score (and thus, the total value) of the completed test-generation work (cf. Tables 6 and 7; right-most x-coordinates match the score values in the tables). The ranking can be read from the plot of the quantile functions from right to left: The right-most data point for VeriFuzz is 1 951, for Klee 1 764, for CoVeriTest 1 524, and so on.

Consumed Resources. One complete test-generation execution of the competition consisted of 21 204 single test-generation runs (see Table 8). The total CPU time was 122 days and the consumed energy 32.1 kWh for one complete competition run for test generation (without validation). Test-suite validation consisted of 21 204 single test-suite validation runs. The total consumed CPU time was 31.1 days. Each tool was executed several times, in order to make sure no installation issues occur during the execution, and thus, the total consumed resources including pre-runs was a multiple of the above-mentioned amounts of resources.

[18] http://www.gnuplot.info

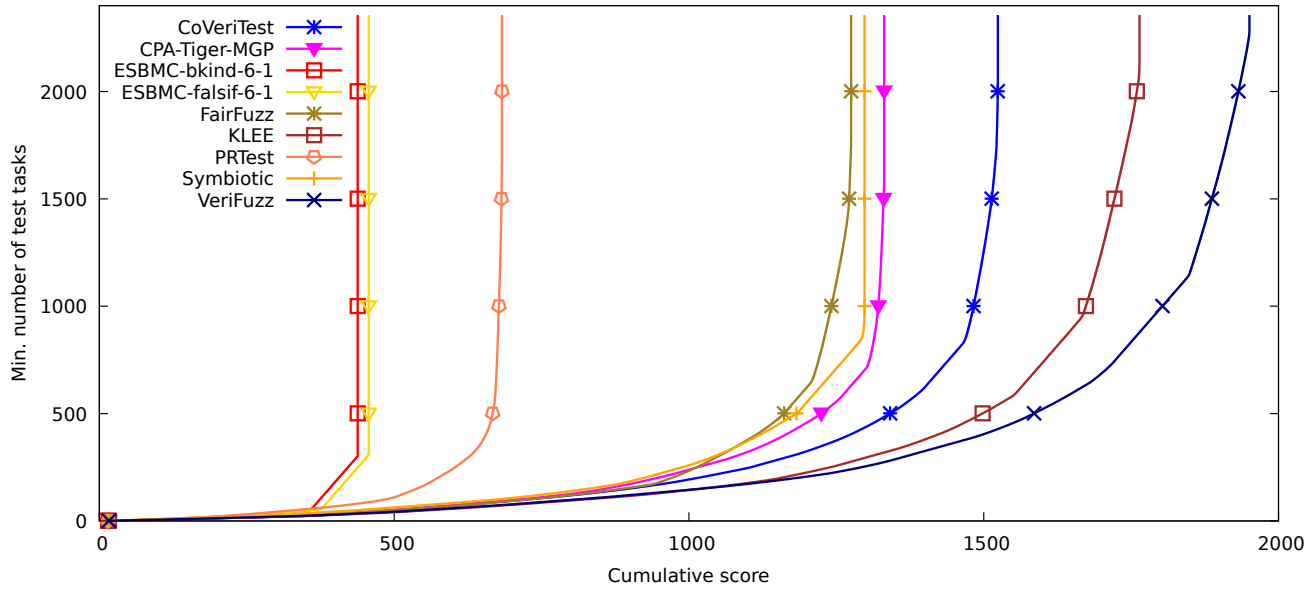

Fig. 8: Quantile functions for category *C-Overall*; each quantile function illustrates the quantile (x-coordinate) of the score points obtained by test-generation runs for a certain minimal number of test-generation tasks (y-coordinate); the graphs are decorated with symbols to make them better distinguishable without color

Table 8: Consumed resources for one Test-Comp 2019 execution (rounded to three significant digits)

Resource	Test-Suite Generation	Test-Suite Validation
Runs	21 204	21 204
CPU Time	122 d	31.1 d
Energy	32.1 kW h	

8 Conclusion and Future Plans

Test-Comp 2019 gave an overview of the state of the art in automatic test generation for C programs. This report describes the organizational aspects of the 1st International Competition on Software Testing (Test-Comp 2019), and the qualitative and quantitative results of the comparative evaluation. The competition attracted nine participating teams from six countries. The feedback from the testing community was positive, and the plan is to hold the competition on software testing annually from now on. We hope that the introduced standards for marking input values, specifying the test-coverage criteria, and writing the generated test suites encourages developers of test generators to apply those standards, in order to deliver tools that are easy to compare and use as components in quality assurance. For the future, the community has plans to increase the number and diversity of the benchmark set, experiment with different time budgets, reduce the size of the generated test suites, include test-generation tasks for Java programs, and extend the categories towards other coverage criteria (e.g., MC/DC) and mutation testing.

Data Availability Statement. The test-generation tasks and results of the competition are published at Zenodo, as described in Table 3. All components and data that are necessary for reproducing the competition are available in public version repositories, as specified in Table 2. Furthermore, the results are presented online on the competition web site for easy access: https://test-comp.sosy-lab.org/2019/results

References

1. Bartocci, E., Beyer, D., Black, P.E., Fedyukovich, G., Garavel, H., Hartmanns, A., Huisman, M., Kordon, F., Nagele, J., Sighireanu, M., Steffen, B., Suda, M., Sutcliffe, G., Weber, T., Yamada, A.: TOOLympics 2019: An overview of competitions in formal methods. In: Proc. TACAS (3). pp. 3–24. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_1
2. Beyer, D.: Competition on software verification (SV-COMP). In: Proc. TACAS. pp. 504–524. LNCS 7214, Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_38
3. Beyer, D.: Second competition on software verification (Summary of SV-COMP 2013). In: Proc. TACAS. pp. 594–609. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_43
4. Beyer, D.: Reliable and reproducible competition results with BenchExec and witnesses (Report on SV-COMP 2016). In: Proc. TACAS. pp. 887–904. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_55
5. Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Proc. TACAS (3). pp. 133–155. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_9
6. Beyer, D.: Competition on software testing (Test-Comp). In: Proc. TACAS (3). pp. 167–175. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_11
7. Beyer, D.: Results of the 1st International Competition on Software Testing (Test-Comp 2019). Zenodo (2020). https://doi.org/10.5281/zenodo.3856661
8. Beyer, D.: SV-Benchmarks: Benchmark set of the 1st Intl. Competition on Software Testing (Test-Comp 2019). Zenodo (2020). https://doi.org/10.5281/zenodo.3856478
9. Beyer, D.: Test suites from Test-Comp 2019 test-generation tools. Zenodo (2020). https://doi.org/10.5281/zenodo.3856669
10. 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
11. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative verifier-based testing. In: Proc. FASE. pp. 389–408. LNCS 11424, Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23
12. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. In: Proc. HVC. pp. 99–114. LNCS 10629, Springer (2017). https://doi.org/10.1007/978-3-319-70389-3_7
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., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
15. 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. pp. 84–99. LNCS 9033, Springer (2015). https://doi.org/10.1007/978-3-662-46675-9_6
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)
17. Cadar, C., Nowack, M.: Klee symbolic execution engine in 2019 (competition contribution). Int. J. Softw. Tools Technol. Transf. (2020). https://doi.org/10.1007/s10009-020-00570-3
18. Chalupa, M., Strejček, J., Vitovská, M.: Joint forces for memory safety checking. In: Proc. SPIN. pp. 115–132. Springer (2018). https://doi.org/10.1007/978-3-319-94111-0_7
19. Chalupa, M., Vitovska, M., Jašek, T., Šimáček, M., Strejček, J.: Symbiotic 6: Generating test-cases by slicing and symbolic execution (competition contribution). Int. J. Softw. Tools Technol. Transf. (2020). https://doi.org/10.1007/s10009-020-00573-0
20. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: VeriFuzz: Program-aware fuzzing (competition contribution). In: Proc. TACAS (3). pp. 244–249. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_22
21. Gadelha, M.R., Menezes, R., Cordeiro, L.: Esbmc 6.1: Automated test-case generation using bounded model checking (competition contribution). Int. J. Softw. Tools Technol. Transf. (2020). https://doi.org/10.1007/s10009-020-00571-2
22. 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 (February 2017). https://doi.org/10.1007/s10009-015-0407-9
23. Godefroid, P., Sen, K.: Combining model checking and testing. In: Handbook of Model Checking, pp. 613–649. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_19
24. Harman, M., Hu, L., Hierons, R.M., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Trans. Software Eng. **30**(1), 3–16 (2004). https://doi.org/10.1109/TSE.2004.1265732
25. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: How did you specify your test suite. In: Proc. ASE. pp. 407–416. ACM (2010). https://doi.org/10.1145/1858996.1859084
26. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D., Păsăreanu, C.S.: Rigorous examination of reactive systems. The RERS challenges 2012 and 2013. Int. J. Softw. Tools Technol. Transfer **16**(5), 457–464 (2014). https://doi.org/10.1007/s10009-014-0337-y
27. Huisman, M., Klebanov, V., Monahan, R.: VerifyThis 2012: A program verification competition. STTT **17**(6), 647–657 (2015). https://doi.org/10.1007/s10009-015-0396-8
28. Jakobs, M.C.: CoVeriTest: Interleaving value and predicate analysis for test-case generation (competition contribution). Int. J. Softw. Tools Technol. Transf. (2020). https://doi.org/10.1007/s10009-020-00572-1
29. Kifetew, F.M., Devroey, X., Rueda, U.: Java unit-testing tool competition: Seventh round. In: Proc. SBST. pp. 15–20. IEEE (2019). https://doi.org/10.1109/SBST.2019.00014
30. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
31. Lemberger, T.: Plain random test generation with PRTest (competition contribution). Int. J. Softw. Tools Technol. Transf. (2020). https://doi.org/10.1007/s10009-020-00568-x
32. Lemieux, C., Sen, K.: FairFuzz-TC: A fuzzer targeting rare branches (competition contribution). Int. J. Softw. Tools Technol. Transf. (2020). https://doi.org/10.1007/s10009-020-00569-w
33. Ruland, S., Lochau, M., Fehse, O., Schürr, A.: CPA/Tiger-MGP: Test-goal set partitioning for efficient multi-goal test-suite generation (competition contribution). Int. J. Softw. Tools Technol. Transf. (2020). https://doi.org/10.1007/s10009-020-00574-z

34. Song, J., Alves-Foss, J.: The DARPA cyber grand challenge: A competitor's perspective, part 2. IEEE Security and Privacy **14**(1), 76–81 (2016). https://doi.org/10.1109/MSP.2016.14
35. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: Proc. IJCAR, pp. 367–373. LNCS 8562, Springer (2014). https://doi.org/10.1007/978-3-319-08587-6_28
36. 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
37. Wendler, P., Beyer, D.: sosy-lab/benchexec: Release 1.18. Zenodo (2019). https://doi.org/10.5281/zenodo.2561835