Proc. TACAS 2019, © Springer

International Competition on Software Testing (Test-Comp)

Dirk Beyer

LMU Munich, Germany

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. Tool competitions have been a driving force for the development of mature tools that represent the state of the art in several research areas. This paper describes the International Competition on Software Testing (Test-Comp), a comparative evaluation of automatic tools for software test generation. Test-Comp 2019 is presented as part of TOOLympics 2019, a satellite event of the conference TACAS.

1 Introduction

Software testing is as old as software development itself, because the easiest way to find out if the 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-case generation. For example, symbolic execution and the idea to use it for test-case generation [14] exists for more than 40 years, efficient implementations (e.g., Klee [8]) had to wait for the availability of mature constraint solvers. On the other hand, with the advent of automatic software model checking, the opportunity to extract test cases from counterexamples arose (see Blast [5] and JPF [16]). In the following years, many techniques from the areas of model checking and program analysis were adapted for the purpose of test-case generation and several strong hybrid combinations have been developed [9].

There are several powerful software test generators available [9], but they are very difficult to compare. For example, a recent study [6] first had to develop a framework that supports to run test-generation tools on the same program source code and to deliver test cases 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 [4] helped to overcome the problem: 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 would also give credits and benefits to students who spend considerable amounts of time developing testing algorithms and software packages (achieving a high rank in the testing competition improves the CV).

Test-Comp is designed to compare automatic state-of-the-art software testers 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 applied to the set of benchmark instances.

Test-Comp uses the benchmarking framework BenchExec [7], which is already successfully used in other competitions, most prominently, all competitions that run on the StarExec infrastructure [15]. 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 goals of Test-Comp are the following:

- Provide a snapshot of the state-of-the-art in software testing to the community. This means to compare, independently from particular paper projects and specific techniques, different test-generation tools in terms of effectiveness and performance.
- 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 students the opportunity to publish about the development work that they have done.
- Establish a set of benchmarks for software testing in the community. This means to create and maintain a set of programs together with coverage criteria, and to make those publicly available for researchers to be used in performance comparisons when evaluating a new technique.
- 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.

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 [3]), (ii) a competition on verifiers with arbitrary environments (RERS [12]), and (iii) a competition on (interactive) verification (VerifyThis [13]). In software testing, there are several competition-like events, for example, the IEEE International Contest on Software Testing, the Software Testing World Cup, and the Israel Software Testing World Cup. 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 is no comparative evaluation of automatic test-generation tools in a controlled environment.

Test-Comp is meant to close this gap. The results of the first edition of Test-Comp will be presented as part of the TOOLympics 2019 event [1], where 16 competitions in the area of formal methods are presented.

2 Organizational Classification

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

- **Automatic:** The tools are executed in a fully automated environment, without any user interaction.
- **Off-site:** The competition takes place independently from a conference location, in order to flexibly allow 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.
- **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 run.

3 Competition Schedule

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

- **Call for Participation:** The organizer announces the competition on the mailing list.[1]
- **Registration of Participation / 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 / Evaluation Phase:** The tool developers submit the final versions of their tool. 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 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.

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

4 Participating Tools

The following tools for automatic software test generation participate in the first edition of Test-Comp (the list provides the tester name, the representing jury member, the affiliation, and the URL of the project web site):

- CoVeriTest, Marie-Christine Jakobs, LMU Munich, Germany
  https://cpachecker.sosy-lab.org/
- CPA/Tiger-MGP, Sebastian Ruland, TU Darmstadt, Germany
  https://www.es.tu-darmstadt.de/testcomp19
- ESBMC-bkind, Rafael Menezes, Federal University of Amazonas, Brazil
  http://www.esbmc.org/
- ESBMC-falsif, Mikhail Gadelha, University of Southampton, UK
  http://www.esbmc.org/
- FairFuzz, Caroline Lemieux, University of California at Berkeley, USA
  https://github.com/carolemieux/afl-rb
- KLEE, Cristian Cadar, Imperial College London, UK
  http://klee.github.io/
- PRTest, Thomas Lemberger, LMU Munich, Germany
  https://github.com/sosy-lab/tbf
- Symbiotic, Martina Vitovská, Masaryk University, Czechia
  https://github.com/staticafi/symbiotic
- VeriFuzz, Raveendra Kumar Medicherla, Tata Consultancy Services, India
  https://www.tcs.com/creating-a-system-of-systems

5 Rules and Definitions

Test Task. A *test task* is a pair of an input program (program under test) and a test specification. A *test run* is a non-interactive execution of a test generator on a single test task, in order to generate a test suite according to the test specification. A *test suite* is a sequence of test cases, given as a directory of files according to the format for exchangeable test-suites.[2]

Execution of a Test Generator. Figure 1 illustrates the process of executing one test generator on the benchmark suite. One test 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 test vectors). The test generator is contributed by the competition participant. The test runs are 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 cases: for bug finding it checks if the bug is exposed and for coverage it reports the coverage using the GNU tool `gcov`.[3]

[2] Test-suite format: https://gitlab.com/sosy-lab/software/test-format/
[3] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

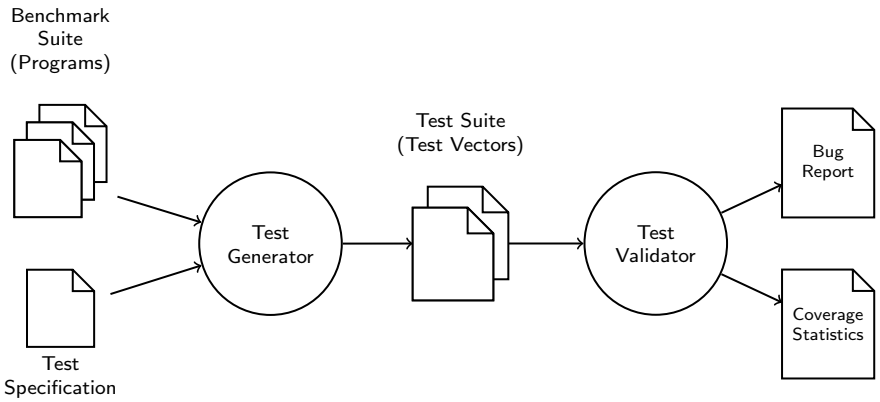

Fig. 1: Flow of the Test-Comp execution for one test generator

Table 1: Coverage specifications used in Test-Comp 2019

Formula	Interpretation
`COVER EDGES(@CALL(__VERIFIER_error))`	The test suite contains at least one test that executes function `__VERIFIER_error`.
`COVER EDGES(@DECISIONEDGE)`	The test suite contains tests such that all branches of the program are executed.

Test Specification. 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 2019).

The definition `init(main())` is used to define the initial states of the program by a call of function `main` (with no parameters). The definition `FQL(f)` specifies that coverage definition `f` should be achieved. The FQL (FShell query language [11]) 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: `COVER( init(main()), FQL(COVER EDGES(@DECISIONEDGE)) )`.

Table 1 lists the two FQL formulas that are used in test specifications of Test-Comp 2019. The first describes a formula that is typically used for bug finding: the test generator should find a test case that executes a certain error function. 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.

Setup. The competition runs on an otherwise unloaded, dedicated compute cluster composed of 168 machines with Intel Xeon E3-1230 v5 CPUs, with 8 processing units each, a frequency of 3.4 GHz, and 33 GB memory. Each test run will be started on such a machine with a GNU/Linux operating system (x86_64-linux, Ubuntu 18.04); there are three resource limits for each test run:

- a memory limit of 15 GB (14.6 GiB) of RAM,
- a runtime limit of 15 min of CPU time, and
- a limit to 8 processing units of a CPU.

Further technical parameters of the competition machines are available in the repository that also contains the benchmark definitions. [4]

License Requirements for Submitted Tester Archives. The testers need to be publicly available for download as binary archive under a license that allows the following (cf. [4]):

- replication 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).

6 Categories and Scoring Schema

Error Coverage. The first category is to show the abilities to discover bugs. The programs in the benchmark set contain programs that contain a bug.

Evaluation by scores and runtime. Every run will be started by a batch script, which produces for every tool and every test task (a C program) one of the following scores:

+1 point:	program under test is executed on all generated test cases and the bug is found (i.e., specified function was called)
0 points:	all other cases

[4] https://gitlab.com/sosy-lab/test-comp/bench-defs/

The participating test-generation tools are ranked according to the sum of points. Tools with the same sum of points are ranked according to success-runtime. The success-runtime for a tool is the total CPU time over all benchmarks for which the tool reported a correct verification result.

Branch Coverage. The second category is to cover as many branches as possible. The coverage criterion was chosen because many test-generation tools support this standard criterion by default. Other coverage criteria can be reduced to branch coverage by transformation [10].

Evaluation by scores and runtime. Every run will be started by a batch script, which produces for every tool and every test task (a C program) the coverage (as reported by `gcov`; value between 0 and 1) of branches of the program that are covered by the generated test cases. The score is the returned coverage.

$+c$ points:	program under test is executed on all generated tests and c is the coverage value as measured with the tool `gcov`
0 points:	all other cases

The participating verification tools are ranked according to the cumulative coverage. Tools with the same coverage are ranked according to success-runtime. The success-runtime for a tool is the total CPU time over all benchmarks for which the tool reported a correct verification result.

7 Benchmark Programs

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 tasks [5], which is also used by SV-COMP [4]. We selected all programs for which the following properties were satisfied (cf. issue on GitHub [6]):

1. compiles with `gcc`, if a harness for the special methods is provided,
2. should contain at least one call to a nondeterministic function,
3. does not rely on nondeterministic pointers,
4. does not have expected result 'false' for property 'termination', and
5. has expected result 'false' for property 'unreach-call' (only for category *Error Coverage*).

This selection yields a total of 2 356 test tasks, namely 636 test tasks for category *Error Coverage* and 1 720 test tasks for category *Code Coverage*. [7] The final set of benchmark programs might be obfuscated in order to avoid overfitting.

[5] https://github.com/sosy-lab/sv-benchmarks
[6] https://github.com/sosy-lab/sv-benchmarks/pull/774
[7] https://test-comp.sosy-lab.org/2019/benchmarks.php

8 Conclusion and Future Plans

This report gave an overview of the organizational aspects of the International Competition on Software Testing (Test-Comp). The competition attracted nine participating teams from six countries. At the time of writing of this article, the execution of the benchmarks of the first edition of Test-Comp was just finished. Unfortunately, the results could not be processed on time for publication. The feedback from the testing community was positive, and the competition on software testing will be held annually from now on. The plan for next year is to extend the competition to more categories of programs and to more tools.

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, part 3. LNCS 11429, Springer (2019)
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.: Software verification with validation of results (Report on SV-COMP 2017). In: Proc. TACAS. pp. 331–349. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_20
4. Beyer, D.: Automatic verification of c and java programs: Sv-comp 2019. In: Proc. TACAS, part 3. LNCS 11429, Springer (2019)
5. 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
6. 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
7. 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
8. 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)
9. 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
10. Harman, M.: We need a testability transformation semantics. In: Proc. SEFM. pp. 3–17. LNCS 10886, Springer (2018). https://doi.org/10.1007/978-3-319-92970-5_1
11. 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
12. 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
13. 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
14. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
15. 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
16. 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