Ultimate TestGen: Test-Case Generation with Automata-based Software Model Checking (Competition Contribution)

Max Barth[1](✉), Daniel Dietsch[2], Matthias Heizmann[2], and Marie-Christine Jakobs[1]

[1] LMU Munich, Munich, Germany
[2] University of Freiburg, Freiburg, Germany
Max.Barth@lmu.de

Abstract. We introduce Ultimate TestGen, a novel tool for automatic test-case generation. Like many other test-case generators, Ultimate TestGen builds on verification technology, i.e., it checks the (un)reachability of test goals and generates test cases from counterexamples. In contrast to existing tools, it applies trace abstraction, an automata-theoretic approach to software model checking, which is implemented in the successful verifier Ultimate Automizer. To avoid that the same test goal is reached again, Ultimate TestGen extends the automata-theoretic model checking approach with error automata.

Keywords: Ultimate Automizer· Test-case generation · Software testing · Test Coverage · Software model checking · Automata

1 Test-Generation Approach

Verification technology has been successfully used in the past to automatically generate test cases [12,14,7,1]. Most existing approaches follow a similar principle. Mainly, they perceive reaching an (uncovered) test goal as a property violation and construct test cases from counterexamples [6]. To build a test suite, they repeatedly check the reachability of still uncovered goals and prove their unreachability or generate test cases from counterexamples that testify the reachability of (uncovered) test goals. To improve the performance of the reachability analysis after detecting the reachability of a test goal, many approaches reuse previous information, e.g., continue the reachability analysis but exclude property violations caused by already covered test goals. Also, our new test-case generator Ultimate TestGen, which is implemented in Java, follows this basic principle.

To analyze the reachability of test goals, Ultimate TestGen relies on trace abstraction [11], an automata-theoretic approach to software model checking, which performs counterexample-guided abstraction refinement (CEGAR) [9] and

[*] Jury Member: Max Barth

D. Beyer and A. Cavalcanti (Eds.): FASE 2024, LNCS 14573, pp. 326–330, 2024.
https://doi.org/10.1007/978-3-031-57259-3_20

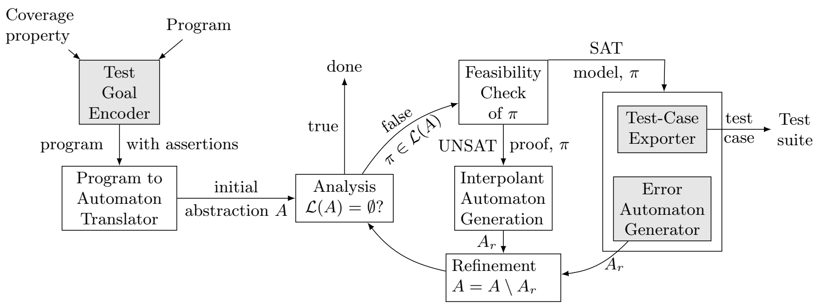

Fig. 1. Overview of the test-case generation approach of Ultimate TestGen

which is implemented in Ultimate Automizer. Figure 1 shows the overview of the test-case generation process performed by Ultimate TestGen. Components highlighted in gray are added to the verification process of Ultimate Automizer and enable test-case generation.

The test-case generation process starts with the encoding of the test goals into the program. To this end, we insert an `assert(false);` statement after each test goal (either a branch or a call to `reach_error()`). Thereafter, we translate the program with the assertions into an automaton A, which becomes the initial abstraction. This initial abstraction represents all possible counterexamples, i.e., the initial automaton accepts a syntactical program path iff it reaches an assert statement (i.e., a violation). Next, we iteratively refine the automaton abstraction until it becomes empty.

If the abstraction still accepts a counterexample path π, we select an arbitrary counterexample path π from the abstraction and check its feasibility. To check the feasibility of π, Ultimate TestGen encodes the path into a formula and checks its satisfiability with an SMT solver. Ultimate TestGen relies on the SMT solvers Z3 [13], CVC4 [3], and MathSAT5 [8]. However, during the check we must ensure that an assert statement introduced to cover an earlier test goal does not prohibit reaching later test goals. Therefore, the feasibility check ignores the assert statements added during test goal encoding.

If the counterexample is spurious, i.e., the formula is unsatisfiable, we use the proof of unsatisfiability to generate an interpolant automaton A_r [10]. The interpolant automaton accepts the counterexample path π and other (counterexample) paths that are infeasible due to a similar reason. We use the interpolant automaton to refine the abstraction and, thus, exclude infeasible paths, which are accepted by the interpolant automaton, from the counterexample search.

If the counterexample is feasible, i.e., the formula is satisfiable, we generate a test case from a model of the formula [6]. To this end, we identify the calls to the `__VERIFIER_nondet` calls and retrieve their values from the model. Then, we export the identified values into a test case in the exchange format[3] used by

[3] https://gitlab.com/sosy-lab/test-comp/test-format/blob/testcomp23/doc/Format.md

Test-Comp [5]. The values are exported in the same order as their corresponding calls occur in the counterexample path π. In addition, we generate an error automaton that accepts all counterexample paths that end in the same test goal as the current counterexample π. We use the error automaton to refine the abstraction and exclude paths from the counterexample search that reach test goals that are already covered.

The last step is the refinement of the abstraction A. This step excludes the paths determined irrelevant because they are known to be infeasible or may not reach uncovered test goals. To this end, we substract the interpolant automaton and error automaton, respectively from the existing abstraction. Hence, each step ensures that the abstraction considered in the next step considers fewer counterexample paths and, thus, guarantees progress of the test-case generation.

2 Discussion of Strengths and Weaknesses

For a comparison of Ultimate TestGen with the other participants of Test-Comp 2024, we refer to the competition report [5].

Ultimate TestGen checks the reachability of every test goal and generates a test case for every goal that it proved reachable. Due to this goal-oriented procedure, it creates relatively small test suites. In addition, if Ultimate TestGen completes the test-case generation process (i.e., result done), we can confidently determine that any test goal not addressed by a test case is indeed unreachable.

Nevertheless, proving the reachability of certain test goals can be hard and requires expensive SMT solver calls. When studying the results for the category `cover-error`, we observe that Ultimate TestGen runs out of resources (time or memory) for many software systems tasks as well as tasks in the categories `XCSP`, `Sequentialized`, `ProductLines`, `ECA`. In addition to the resource issue, we observe that sometimes our tests are not confirmed by the validator, which seems to be a bug of the translation of the counterexamples into the test cases. Still, there also exist categories like `loops`, `heap`, `arrays`, and `fuzzle` in which Ultimate TestGen performs rather well.

Looking at the `cover-branches` category, we observe that for many software systems tasks as well as for certain float tasks, we already fail to construct the automaton from the program because required C features are yet not supported by the program to automaton translation. In these cases, the test-case generation procedure does not even start. In addition, Ultimate TestGen has problems in detecting the feasibility of error traces for Linux device driver tasks because large string literals are not precisely encoded. For other task categories like `AWS`, `Sequentialized`, `ProductLines`, `Hardware`, `Fuzzle`, `ECA`, and `Combinations`, we observe that reaching the test goals is expensive and Ultimate TestGen runs out of resources (time, memory) before covering a significant amount of test goals. While we have seen the resource issue for the `cover-error` category, too, the `Hardness` tasks reveal another issue with our test-case exporter, which makes Ultimate TestGen crash. The reason for the crash is that our test-case exporter failed to translate values from the SMT-LIB [2] FloatingPoint format back to certain C types such as ulong. Note that the C types float and double were not an issue. Still, there exist task categories like e.g., `loops`, `control-flow`, `bitvectors`, or `XCSP` for which Ultimate TestGen performs well and achieves high coverage values.

3 Setup and Configuration

Ultimate TestGen is part of the Ultimate framework[4], which is licensed under LGPLv3. To execute Ultimate TestGen in the version submitted to Test-Comp 2024 [4], one requires Java 11 and Python 3.6 and must invoke the following command.

```
./Ultimate.py -spec <p> -file <f> -architecture <a> -full-output
```

where `<p>` is a Test-Comp property file, `<f>` is an input C file, and `<a>` is the architecture (`32bit` or `64bit`). During execution of the command, the generated tests are saved as `.xml` files in the exchange format for test cases required by Test-Comp [5]. In Test-Comp 2024, we use the above command to participate with Ultimate TestGen in both Test-Comp categories: `cover-error` (i.e., bug finding by covering the call to `reach_error`) and `cover-branches` (i.e., code coverage).

Data Availability The Test-Comp 2024 version of Ultimate TestGen is available online on Zenodo [4] and on GitHub[5]. Its corresponding benchmark definition file is available on GitLab[6].

References

1. 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. pp. 324–329. LNCS 13991, Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_18
2. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)
3. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proc. CAV. pp. 171–177. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Barth, M., Dietsch, D., Heizmann, M., Jakobs, M.C.: Ultimate TestGen. Zenodo (2023), https://doi.org/10.5281/zenodo.10071568
5. Beyer, D.: Automatic testing of C programs: Test-Comp 2024. Springer (2024)

[4] https://ultimate.informatik.uni-freiburg.de and github.com/ultimate-pa/ultimate
[5] https://github.com/ultimate-pa/ultimate/tree/ea2a3342b0e9ae9c8710d9bc5a32ec c16b7297dd
[6] https://gitlab.com/sosy-lab/test-comp/bench-defs/-/blob/main/benchmark-defs/ utestgen.xml

6. Beyer, D., Chlipala, A., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE. pp. 326–335. IEEE (2004). https://doi.org/10.1109/ICSE.2004.1317455
7. Chalupa, M., Novák, J., Strejček, J.: Symbiotic 8: Parallel and targeted test generation (competition contribution). In: Proc. FASE. pp. 368–372. LNCS 12649, Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_20
8. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS. pp. 93–107. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
9. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. CAV. pp. 154–169. LNCS 1855, Springer (2000). https://doi.org/10.1007/10722167_15
10. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Proc. SAS. pp. 69–85. LNCS 5673, Springer (2009). https://doi.org/10.1007/978-3-642-03237-0_7
11. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
12. Jakobs, M.C.: CoVeriTest with dynamic partitioning of the iteration time limit (competition contribution). In: Proc. FASE. pp. 540–544. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_30
13. de Moura, L.M., Bjørner, N.S.: Z3: An efficient SMT solver. In: Proc. TACAS. pp. 337–340. LNCS 4963, Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
14. 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. pp. 520–524. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_26