International Journal on Software Tools for Technology Transfer (2021) 23:847–851
https://doi.org/10.1007/s10009-020-00572-1

COMPETITIONS AND CHALLENGES

Special Issue: TestComp 2019

CoVeriTest: interleaving value and predicate analysis for test-case generation

Marie-Christine Jakobs[1,2]

Published online: 6 July 2020

Abstract
Verification techniques are well-suited for automatic test-case generation. They basically need to check the reachability of every test goal and generate test cases for all reachable goals. This is also the basic idea of our CoVeriTest submission. However, the set of test goals is not fixed in CoVeriTest, instead we can configure the set of test goals. For Test-Comp'19, we support the set of all `__VERIFIER_error()` calls as well as the set of all branches. Thus, we can deal with the two test specifications considered in Test-Comp'19. Since the tasks in Test-Comp are diverse and verification techniques have different strengths and weaknesses, we also do not stick to a single verification technique, but use a hybrid approach that combines multiple techniques. More concrete, CoVeriTest interleaves different verification techniques and allows to configure the cooperation (i.e., information exchange and time limits). To choose from a large set of verification techniques, CoVeriTest is integrated into the analysis framework CPAchecker. For the competition, we interleave CPAchecker's value and predicate analysis and let both analyses resume their analysis performed in the previous iteration.

Keywords Test-case generation · Software testing · Model checking · Cooperative verification · CPAchecker

1 Test-generation approach

It is well known that test-case generation approaches come with different strengths and weaknesses, i.e., they are well-suited for certain programs and perform poorly on others. The programs in the Test-Comp benchmark set are diverse. No single test-case generation approach will perform well on all of them. To deal well with all Test-Comp tasks, we thus need to use different test-case generation approaches. Therefore, our Test-Comp'19 submission CoVeriTest is a hybrid approach that combines different approaches. Encouraged by recent advances in software verification, verifiers' bug finding capabilities as well as their abilities to achieve high coverage—each test goal is encoded as one reachability query–, we combine different verification techniques. More concrete, we use a combination that is one specific instance of **co**operative, **veri**fier-based **test**ing [4]. Co-operative, verifier-based testing iteratively combines different verification techniques for test-case generation. In each iteration, it runs the verification techniques in sequence limiting each technique to its individual, user-defined time budget. Additionally, one can define the level of cooperation, i.e., which information is exchanged between different verification runs. Currently, the exchange of precisions, abstract reachability graphs, and conditions [2], which describe program paths that have already been explored, is supported.

For our CoVeriTest submission, we select the instance of cooperative, verifier-based testing that performed best in a recent study [4]. It iteratively combines two verification techniques: a value analysis [7] and a predicate analysis [6]. The value analysis explicitly tracks the values of all variables stored in its precision. For all remaining variables, the analysis assumes that they can have any possible value. The predicate analysis uses predicate abstraction with adjustable block encoding [6], which is configured to abstract at loop heads only. Both analyses use counterexample-guided abstraction refinement [9] to adapt their precision (the set of tracked variables or the set of predicates). Furthermore, cooperation between verification runs is configured to

Marie-Christine Jakobs: Jury-member.

✉ Marie-Christine Jakobs
jakobs@sosy.ifi.lmu.de

1 LMU Munich, Munich, Germany

2 Present Address: TU Darmstadt, Darmstadt, Germany

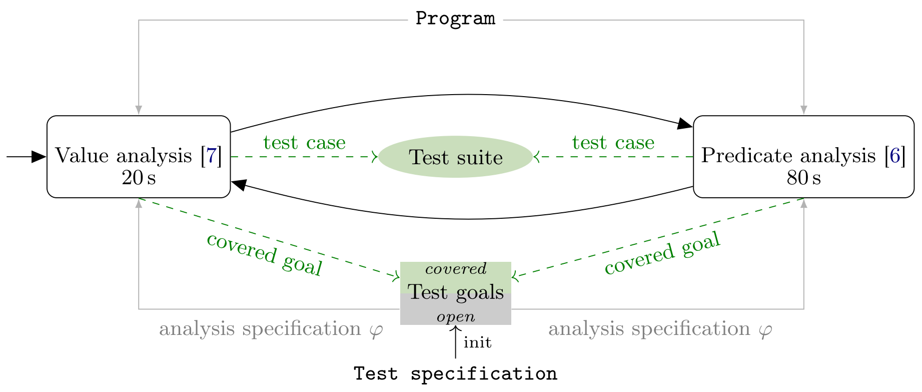

Fig. 1 CoVeriTest workflow for Test-Comp'19

exchange abstract reachability graphs between different verification runs of the same analysis. Practically, this means that verification runs either resume a previous value or predicate analysis.

Figure 1 presents the workflow of our CoVeriTest submission. Given a program and a test specification, it first translates the test specification (either cover call to `__VERIFIER_error()` or cover all branches) into a set of test goals. During test-case generation, the set of test goals is split into two disjoint sets: *open* (goals that still need to be considered) and *covered* (goals for which a test case has been generated). Initially, the set of covered goals is empty. Both analyses, i.e., the value and predicate analysis, use the set of open test goals as specification, which must not be reached.

In each round, our CoVeriTest submission first performs test-case generation with the value analysis and thereafter with the predicate analysis. The limit of the value analysis is 20 s per round, and the limit of the predicate analysis is 80 s. Both analyses resume their exploration from the previous round and do not exchange any further information. Note that for the sake of readability, the figure does not show the exchange of the abstract reachability graph required for analysis resumption. The time limits and the cooperation setting (reuse own results, but do not exchange information between different analyses except for covered goals) are the result of a bunch of experiments with the Test-Comp tasks.

Next, let us look at the test-case generation details. Whenever one of the two analyses reports a violation of the specification, a test case is constructed from the corresponding counterexample [1]. The test case is output in the Test-Comp exchange format.[1] To avoid that another test case is generated for the test goal that triggered the specification violation, this test goal is moved from the set of open goals to the set of covered goals. Thereafter, the analysis that reported the violation is continued with the modified specification.[2]

2 Tool architecture

CoVeriTest is part of CPAchecker [5], a software analysis tool mainly written in Java. In its front end, CPAchecker uses the Eclipse CDT parser.[3] In addition, CPAchecker employs JavaSMT [10] to integrate different SMT solvers. For Test-Comp'19, we choose CPAchecker's default SMT solver MathSAT5 [8].

The core of CPAchecker is the configurable program analysis (CPA) framework [3]. The CPA framework provides the basis to express different verification approaches and consists of two parts: configurable program analyses (CPAs) and the CPA algorithm. CPAs define program analyses (abstract domain plus the analysis operators post, merge, and stop) and can freely be combined to build more complex analyses. Two example CPAs are the value and predicate analysis used by CoVeriTest. Given a CPA and a program, the CPA algorithm performs the corresponding reachability analysis.

On top of the CPA framework, different verification algorithms, e.g., counterexample-guided abstraction refinement (CEGAR) [9], are integrated into CPAchecker. Also, CoVeriTest is implemented on top of the CPA framework. Basically, we added two algorithms: a test-case generation algorithm and a circular algorithm performing the continuous iteration over a set of analyses. The test-case generation algorithm wraps another analysis and is responsible for test-case generation. To this end, it runs the wrapped analysis, constructs test cases from counterexamples [1] reported by the wrapped analysis, updates the analysis specification (removing covered goals), and resumes the wrapped analysis afterward. The circular algorithm instance that we use for CoVeriTest in Test-Comp'19 iterates over two instances of the test-case generation algorithm, one wrapping the predicate analysis and the other the value analysis.

[1] https://gitlab.com/sosy-lab/software/test-format/tree/master

[2] The set of open test goals changed.

[3] https://www.eclipse.org/cdt/

3 Strengths and weaknesses

CoVeriTest won the third place in Test-Comp'19 in both categories, Cover-Error and Cover-Branches, as well as in the category Overall. Two main characteristics of CoVeriTest are responsible for its success. To tackle the diverse set of tasks in the Test-Comp benchmark set, CoVeriTest combines different analyses for test-case generation. CoVeriTest's specialty is the iterative combination. Additionally, CoVeriTest's test-case generation is directed at the test specification of interest. Since CoVeriTest is based on verification technologies, it is easy for CoVeriTest to specifically search for the test goals of interest and abstract away from unimportant program behavior. This is, for example, reflected in the low number of test cases (typically less than one) produced for the bug finding category Cover-Error. Moreover, most of the produced tests in this category are confirmed. Only in a few cases, we failed to properly generate a test case from the counterexample. Hence, we are rather confident that we typically produce valuable test cases.

Looking at the detailed results,[4] we observe that CoVeriTest performs well on the subcategories BitVectors, Control-Flow, Floats, Heap, and Loops.

Furthermore, we observe that CoVeriTest has difficulties with subcategory Arrays and ECA. For us, this is no surprise. We already know that the underlying analyses have problems with tasks that contain large arrays. For the ECA tasks, we identified two possible problems. First, the branching structure of the ECA tasks makes them hard for CoVeriTest. On the one hand, the branch conditions contain many Boolean connectors. Since CPAchecker internally splits Boolean connections before CoVeriTest computes its set of test goals, the considered set of test goals is much larger than the actual number of branches. On the other hand, only few syntactic paths are feasible in each loop iteration of an ECA task. CoVeriTest must exclude many infeasible paths in each loop iteration, which is costly. Second, we know that the value analysis is much better on the ECA tasks than the predicate analysis, which gets more runtime in our configuration. An adaptive division of the runtime might improve the performance on the ECA tasks.

Surprisingly, the performance of CoVeriTest differs for the subcategories Recursive and Sequentialized when it comes to covering errors and branches, respectively. CoVeriTest has significant problems with the subcategory Recursive and problems with about 40% of tasks in the subcategory Sequentialized (especially, lcr tasks and overflows) when the task is to cover the error. Since we know that our analyses do not support recursion, we expect a bad performance for recursive tasks. Interestingly, many branches of the recursive tasks can be reached without a recursive call. Moreover, the error in some of the sequentialized tasks seems to be particularly difficult to reach in comparison with the rest of the reachable branches.

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

Table 1 Comparing the overall scores achieved by CoVeriTest submitted to Test-Comp with the scores achieved by a sequential analysis combination achieved on the Test-Comp benchmark set

Property	Score Test-Comp	Score Sequential
Cover-Error	331	360
Cover-Branches	753	744

At last, we want to show that the iterative combination of analyses in CoVeriTest pays off. Our motivation for an iterative combination is that analyses getting stuck while trying to cover a particular goal can recover after another analysis in the combination covered the goal. To study whether CoVeriTest benefits from this idea, we compare CoVeriTest with a sequential combination of its analyses. We derive the sequential combination from the iterative CoVeriTest configuration by only changing the time limits. The new, sequential time limits for the value and predicate analysis are 180 s and 720 s (the accumulated time spent by each analysis in the original CoVeriTest configuration). Then, we compare the results of both configuration on the Test-Comp benchmark set.

Table 1 shows the sum of all scores achieved by both configurations in each of the two Test-Comp categories Cover-Error and Cover-Branches. We observe that the CoVeriTest configuration submitted to Test-Comp, which uses an interleaved combination, covers less errors. All errors exclusively covered by the sequential combination are found by the value analysis. We think the sequential combination performs better for those tasks because some operation of the value analysis, e.g., counterexample check or refinement, cannot be finished within 20 s (time limit for value analysis in each iteration in the Test-Comp configuration). However, it can be performed within the 180 s time limit (time limit of value analysis in the sequential combination). Nevertheless, the Test-Comp configuration detects one error that the sequential combination does not report. For branch coverage, the original intention of CoVeriTest, the picture looks different. When using an inter-leaved combination (Test-Comp configuration), a higher overall score is achieved. Interleaving pays off. To further substantiate this, let us look at the scatter plot in Fig. 2. For each Test-Comp task in the category Cover-Branches, the scatter plot compares the branch coverage of the Test-Comp submission (x axis) with the coverage of the sequential combination (y axis). We observe that many points are on the diagonal, few are in the upper right half, and a significant amount of points is in the lower right half (meaning the CoVeriTest configuration

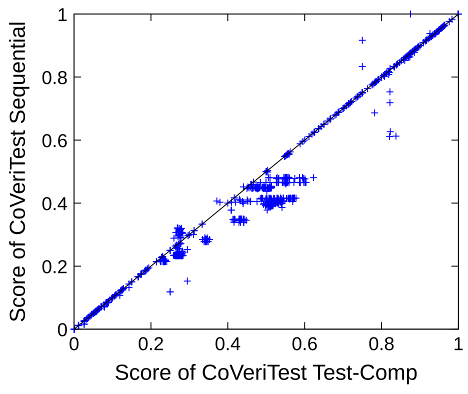

Fig. 2 Scatter plot that compares the branch coverage (score) of our CoVeriTest submission and of the sequential analysis combination

submitted to Test-Comp performs better). Hence, when generating test-suites for structural coverage properties, an interleaved combination—as used for Test-Comp—pays off.

4 Setup and configuration

To set up CoVeriTest, one needs to download CPAchecker[5] (the tool in which CoVeriTest is integrated). For Test-Comp'19, we used the revision 30375 from the CPAchecker trunk. Furthermore, one requires a Java 8 runtime environment. After proper setup, the following command executes CoVeriTest on a single program `program.i`. The file `property.prp` is a placeholder for the test specification, either `coverage-error-call.prp` or `coverage-branches.prp`.

```
scripts/cpa.sh -testcomp19 -benchmark
  -heap 10000m —spec property.prp program.i
```

The command above assumes that `program.i` runs in a 32-bit environment. For C programs requiring a 64-bit environment, one must add the parameter `-64`. Moreover, for machines with less RAM one can adjust the amount of memory given to the Java VM. Just change the memory value passed with the parameter `-heap`.

During execution, CoVeriTest produces a test-suite, which consists of a metadata file and test-case files in the XML format[6] defined by Test-Comp. CoVeriTest writes the test-suite to a directory named `test-suite`. This directory is a subdirectory within the output directory of CPAchecker.

[5] https://cpachecker.sosy-lab.org

[6] https://gitlab.com/sosy-lab/software/test-format/tree/master

We participate with our CoVeriTest submission in all categories of Test-Comp'19.

5 Project and contributors

CoVeriTest is integrated into the open-source project CPAchecker, which is maintained by Dirk Beyer and his group at LMU Munich. Currently, members of the Institute for System Programming of the Russian Academy of Sciences, Paderborn University, TU Darmstadt, and several other universities and institutes contribute to or use CPAchecker. We would like to thank all contributors for their work on CPAchecker.

Acknowledgements Open Access funding provided by Projekt DEAL.

References

1. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proceedings of ICSE, pp. 326–335. IEEE (2004). https://doi.org/10.1109/ICSE.2004.1317455
2. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: a technique to pass information between verifiers. In: Proceedings of FSE. ACM (2012). https://doi.org/10.1145/2393596.2393664
3. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In: Proceedings of CAV, LNCS 4590, pp. 504–518. Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_51
4. Beyer, D., Jakobs, M.: CoVeriTest: cooperative verifier-based testing. In: Proceedings of FASE, LNCS 11424, pp. 389–408. Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23
5. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Proceedings of CAV, LNCS 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
6. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proceedings of FMCAD, pp. 189–197. FMCAD (2010). http://ieeexplore.ieee.org/document/5770949/
7. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proceedings of FASE, LNCS

7793, pp. 146–162. Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_11

8. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proceedings of TACAS, LNCS 7795, pp. 93–107. 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 for symbolic model checking. J. ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643
10. Karpenkov, E.G., Friedberger, K., Beyer, D.: JavaSMT: a unified interface for SMT solvers in Java. In: Proceedings of VSTTE, LNCS 9971, pp. 139–148. Springer (2016). https://doi.org/10.1007/978-3-319-48869-1_11