International Journal on Software Tools for Technology Transfer (2021) 23:853–856
https://doi.org/10.1007/s10009-020-00574-z

COMPETITIONS AND CHALLENGES

Special Issue: TestComp 2019

CPA/Tiger-MGP: test-goal set partitioning for efficient multi-goal test-suite generation

Sebastian Ruland[1] · Malte Lochau[1] · Oliver Fehse[1] · Andy Schürr[1]

Published online: 3 June 2020

Abstract
Software model checkers can be used to generate high-quality test cases from counterexamples of a reachability analysis. However, naïvely invoking a software model checker for each test goal in isolation does not scale to large programs as a repeated construction of an abstract program model is expensive. In contrast, invoking a software model checker for reaching all test goals in a single run leads to few abstraction possibilities and thus to low scalability. Therefore, our approach pursues a test-suite generation technique that incorporates configurable multi-goal set partitioning (MGP) including configurable partitioning strategies and simultaneous processing of multiple test goals in one reachability analysis. Our approach employs recent techniques from multi-property verification in order to control the computational overhead for tracking multi-goal reachability information. Our tool, called CPA/Tiger-MGP, uses predicate-abstraction-based program analysis in the model-checking framework CPAchecker.

Keywords CPAchecker · Test-goal partitioning · Multi-goal test coverage

1 Software architecture

Our tool CPA/Tiger-MGP is implemented in Java and is based on the CPAchecker framework [3]. The CPAchecker framework for software verification is based on Configurable Program Analysis (CPA) [2] and allows developers to easily integrate new program-analysis techniques by implementing additional CPAs. A CPA defines a program analysis technique consisting of an abstract domain and the analysis operators *post*, *merge* and *stop*. All CPAs can be used in any combination with other CPAs in a single analysis, by specifying the corresponding configuration parameters. Additionally, different verification algorithms are implemented in CPAchecker, such as counterexample-guided abstraction refinement (CEGAR), which enables dynamic adaptation of precision (i.e., the precision of the abstract representation) during runtime [6]. Our tool CPA/Tiger-MGP uses predicate analysis as main CPA, which uses in our configuration MathSAT5 as SMT Solver [5]. The predicate analysis uses predicate abstraction with adjustable block encoding [4]. Additionally, CPAchecker provides us with a built-in parser for C programs using the Eclipse CDT project.[1]

Our automated multi-goal test-generation algorithm CPA/Tiger-MGP is implemented on top of the CPAchecker framework. CPA/Tiger-MGP is able to derive sets of test goals from input programs with respect to a given test-goal specification and to repeatedly execute CPAchecker reachability analysis queries until every reachable test goal is covered by at least one test case. CPAchecker is originally used to verify correctness properties (e.g., non-reachability of error locations) of input programs. If such a property is violated, a witness (i.e., counterexample) is returned, otherwise the return value is *true*. CPA/Tiger-MGP uses this representation by encoding each test goal into a respective (non-)reachability property (e.g., a test goal corresponding

✉ Sebastian Ruland
sebastian.ruland@es.tu-darmstadt.de

Malte Lochau
malte.lochau@es.tu-darmstadt.de

Andy Schürr
andy.schuerr@es.tu-darmstadt.de

[1] Department of Electrical Engineering and Information Technology, Real-Time Systems Lab, Technical University of Darmstadt, Darmstadt, Germany

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

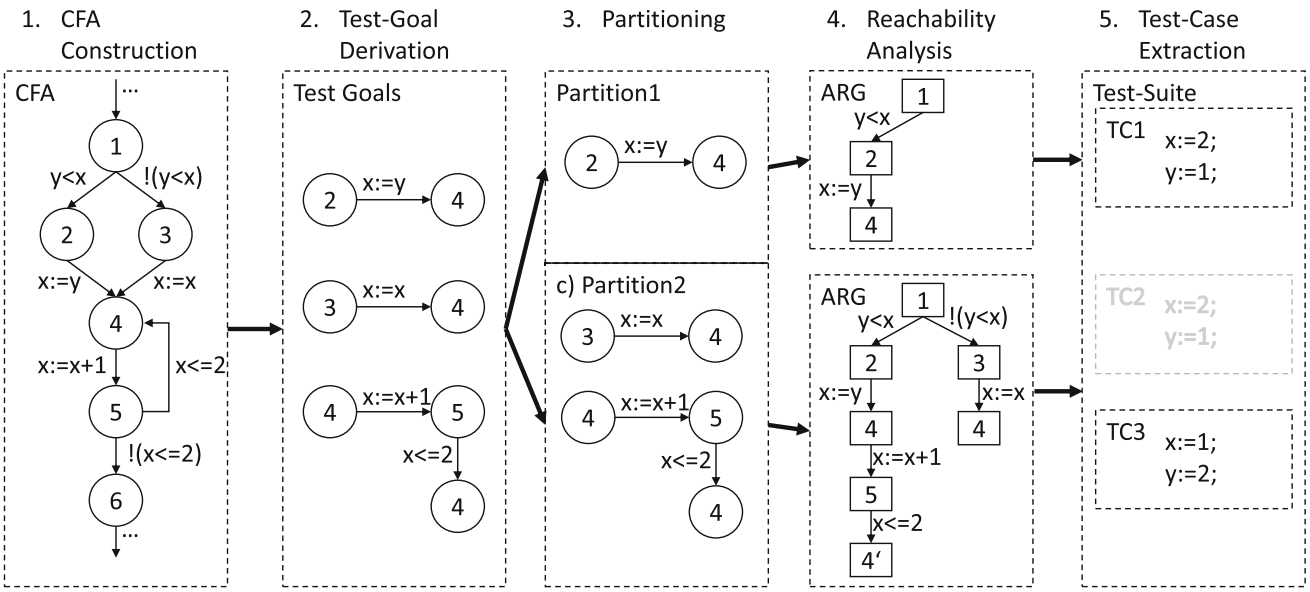

Fig. 1 Overview of CPA/Tiger-MGP

```
1   ...
2   if(y<x) {
3     x = y;
4   } else {
5     x = x;
6   }
7   do {
8     x++;
9   } while(x<=2)
10  ...
```

Fig. 2 C Code Example

to a particular line of code of the input program is converted into a property requiring non-reachability of this program location in all possible program-execution paths). If this property is violated (e. g., the line of code is reachable by a valid program-execution path), CPACHECKER returns this execution as a counterexample from which CPA/Tiger-MGP extracts a test case consisting of a vector of input-variable value assignments together with an (optional) output value for those inputs. Otherwise, if the property holds, the test goal is infeasible (i.e., there exists no valid test case being able to reach the test goal) [1].

2 Test-generation approach

CPA/Tiger-MGP uses as input a C program under test and transforms the program into a so-called control-flow automaton (in the following referred to as CFA), as well as a property specification (i. e., a description of the test goals to be reached within the resulting CFA representation). The CFA is a labeled graph whose nodes correspond to program locations and whose edges correspond to control flows between program locations, being labeled with respective assumptions and assignments over program variables according to the data flows in the program. An example of a CFA representation is provided in Fig. 1 on the left, which corresponds to the C program source code in Fig. 2.

In CPA/Tiger-MGP, each test-goal description consists of a list of CFA edges, which must all be reached by a test case in order to satisfy that goal. During program analysis, the CFA is traversed and an abstract reachability graph (ARG) is built by all CPAs selected by the current configuration. In our approach, we use predicate analysis as main CPA. To keep track of the test goals, we implemented an additional CPA called multi-goal CPA. This CPA contains the set of test goals to be covered. For each edge traversed during reachability analysis, the multi-goal CPA checks for each goal if this edge is the next edge required for covering the goal. If a test goal is completely covered after a traversed edge, a counterexample for its non-reachability can be constructed. If the counterexample is not spurious (i. e., it corresponds to a valid program-execution path), a test case by means of input-variable value assignment can be derived from the corresponding path condition. Furthermore, the test goal is removed from the set of remaining goals and the reachability analysis continues for the remaining test goals. Otherwise, a further CEGAR iteration will be invoked to further refine the abstract representation and the reachability analysis is restarted on the refined ARG.

Since CEGAR has to be performed for multiple goals, the more goals we process simultaneously, the more information we have to track during reachability analysis. Additionally, if a test goal comprises multiple CFA edges (e. g., Goal 3 in

the second block of Fig. 1), we must track the current state of all not-yet-covered goals, which might become intractable in case of larger programs and/or larger sets of test goals. To tackle this problem, we allow to divide the overall set of test goals into smaller subsets, called *partitions*. The main idea of the CPA/Tiger-MGP algorithm is illustrated in Fig. 1.

1. The CFA representation is created for the input program,
2. the list of test goals is derived from the CFA according to the given test-goal specification,
3. the test goals are assigned to partitions depending on the selected partitioning strategy,
4. for each partition, a multi-goal reachability analysis is performed by constructing the respective ARG,
5. for each successfully reached test goal, a counterexample is generated, from which a test case (i. e., program inputs corresponding to the program path of the counterexample) is extracted, and
6. finally, all test cases are integrated into a single test suite, where redundant test cases should be removed (e. g., TC2 in Fig. 1).

CPA/Tiger-MGP supports various strategies for multi-goal set partitioning. First, it is possible to use partitions with either random or fixed sizes with either a fixed or random number of partitions. Additionally, it is possible to employ further context information from the CFA to select test goals, which are more likely to be assigned to the same partition (e. g., test goals sharing a particular fraction of CFA paths). We evaluated the performance of different strategies and observed for the Test-Comp benchmarks that dividing the set of test goals into four subsets with randomly distributed test goals leads to the best results on average. For each of these partitions, we run a reachability analysis and try to cover as many test goals of the partition as possible as described above. For every counterexample found for a particular goal, we further check if this counterexample additionally covers other not-yet-discovered goals from the current partition as well as every other not-yet-processed partition. If so, those test goals can also be removed from the respective partitions. For each analysis of a single partition, we grant a limited amount of CPU time, given by the timeout value configured for the whole analysis divided by the number of partitions. To keep track of analysis timeouts, we additionally implemented a timeout CPA. In this way, we are able to manually terminate individual analysis runs without the need to terminate the whole test-suite generation process altogether.

3 Strengths and weaknesses

The multi-goal analysis potentially leads to a significantly reduced number of reachability analysis runs as compared to a goal-by-goal approach. Despite the additional CPU time needed for a single analysis, the reduced amount of analysis runs provides a considerable trade-off in terms of total CPU time. Additionally, using test-goal set partitioning further increases the performance of CPA/Tiger-MGP, as compared to analyzing all test goals simultaneously.

One weakness of CPA/Tiger-MGP is the current restriction of using predicate analysis as main CPA as for some input programs, explicit-value analysis would perform significantly better. Recent research has shown that a combination of multiple configurations into a single analysis leads to further performance improvements [7]. Additionally, CPA/Tiger-MGP is currently not explicitly tailored to handle recursion thus potentially leading to poor performance or even no results at all in case of recursive programs. CPA/Tiger-MGP is, by intention, only able to play its full strength if applied to programs with multiple test goals. Therefore, for tasks such as finding single error location in a program, CPA/Tiger-MGP will not be able to apply any partitioning strategy and therefore not being able to achieve any increase in performance as compared to processing single goals per reachability analysis.

Results As expected, CPA/Tiger-MGP performs better on programs consisting of many test goals during Test-Comp19. CPA/Tiger-MGP managed to reach the 4th rank in the *Code Coverage* category while reaching the 6th place in the *Finding Bugs* category. In contrast, CPA/Tiger-MGP did not perform well on event-condition-action (ECA) programs which consist of large amounts of variables and branches thus being quite expensive for a predicate-based abstraction analysis.

4 Setup and configuration

The submitted version of CPA/Tiger-MGP is built from revision 30601 from the official CPAchecker repository, branch tigerIntegration2.[2] Additionally, the submitted version is archived at https://gitlab.com/sosy-lab/test-comp/archives-2019.

To run CPA/Tiger-MGP on a single file, enter the following command:

```
1  scripts/cpa.sh -benchmark -heap 10000M -tigertestcomp19
       -spec spec.prp task
```

where *spec* is either the path to the *coverage-error-call* or *coverage-branches* property file and task specifies the C file or the intermediate file to be analyzed. CPA/Tiger-MGP

[2] https://svn.sosy-lab.org/software/cpachecker/branches/tigerIntegration2

prints statistics of the run on the console and writes meta-data on generated test cases and test suites to the output folder. To reproduce our results, use a Linux system with Java 8 runtime environment, BenchExec[3] and SV-Benchmarks,[4] and run Benchexec with the following files:

- the benchmark definition *cpa-tiger.xml* which is archived at https://gitlab.com/sosy-lab/test-comp/bench-defs/tree/master/benchmark-defs, and
- the tool-info module *cpachecker.py* which can be found at https://github.com/sosy-lab/benchexec/tree/master/benchexec/tools.

CPA/Tiger-MGP has participated in the categories *Finding Bugs* and *Code Coverage*.

5 Project and contributors

CPAchecker is an open-source project for software verification maintained by the Software Systems Lab from the Ludwig-Maximilian University in Munich. The developers are members of an international research group from Ludwig-Maximilian University of Munich, the University of Passau, the TU Darmstadt and the Institute for System Programming of the Russian Academy of Sciences. More information about CPAchecker can be found at https://cpachecker.sosy-lab.org/.

Acknowledgements Open Access funding provided by Projekt DEAL. This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

[3] https://github.com/sosy-lab/benchexec

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

References

1. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proceedings of the 26th International Conference on Software Engineering. pp. 326–335. International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA. http://dl.acm.org/citation.cfm?id=998675.999437 (2004)
2. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification, pp. 504–518. Springer, Berlin (2007)
3. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification, pp. 184–190. Springer, Berlin (2011)
4. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design. pp. 189–198. FMCAD '10, FMCAD Inc, Austin, TX. http://dl.acm.org/citation.cfm?id=1998496.1998532 (2010)
5. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 93–107. Springer, Berlin (2013)
6. Clarke, E., 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
7. Löwe, S., Mandrykin, M., Wendler, P.: CPAchecker with sequential combination of explicit-value analyses and predicate analyses. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 392–394. Springer, Berlin (2014)