SCIENCE CHINA
Information Sciences

• INSIGHT •

September 2016, Vol. 59 098101:1–098101:2
doi: 10.1007/s11432-016-5589-6

Automated coverage-driven testing: combining symbolic execution and model checking

Ting SU[1], Geguang PU[1]*, Weikai MIAO[1]*, Jifeng HE[1] & Zhendong SU[2]

[1]*Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai* 200062*, China;*
[2]*Department of Computer Science, University of California, Davis* 95616*, USA*

Received May 9, 2016; accepted June 4, 2016; published online August 23, 2016

Citation Su T, Pu G G, Miao W K, et al. Automated coverage-driven testing: combining symbolic execution and model checking. Sci China Inf Sci, 2016, 59(9): 098101, doi: 10.1007/s11432-016-5589-6

Software testing is the primary way to ensure software quality, but occupies more than 50% the cost of software development [1]. It was estimated that software failures cost the US economy alone about 60 billion each year largely due to inadequate software testing infrastructures [2].

In software industry, white-box testing is one testing method to exercise internal structures of programs. The testers choose appropriate test inputs to exercise program paths in the code and verify the outputs. However, manually conducting this task has several challenges: (1) Identifying inputs is very time-consuming due to the heavy computation of path conditions from a large number of paths; (2) Different code coverage criteria, e.g., statement, branch, logical [3], and data flow coverage criteria [4], are required to be enforced in different scenarios; (3) Infeasible test objectives (the paths from the program entry to such objectives are unexecutable) further undermine the effectiveness of coverage-driven testing.

The aforementioned challenges underline the importance of automating white-box testing. To this end, we propose a novel testing approach, which combines two state-of-the-art techniques, i.e., dynamic symbolic execution [5] and software model checking [6], to automate coverage-driven white-box testing. The key insight is to reduce the problem of finding testing inputs for test objectives (e.g., statements, branches, or def-use pairs) to the problem of finding program paths to reach them. At the high level, given the program and the intended coverage criterion, our approach (1) identifies all test objectives w.r.t. the coverage criterion, (2) outputs the test inputs to exercise them, and (3) eliminates infeasible ones — without any false positives.

In white-box testing, statement and branch coverage answer whether each statement and each branch of a conditional statement have been executed, respectively. Data-flow coverage [4] aims to cover each pair of a variable definition and its corresponding use. To achieve more efficient coverage-driven testing, we introduce three key parts of our approach in the following, i.e., (1) guided symbolic execution, (2) adapted model checking, and (3) combining them together. Here we represent a program path as a sequence of control points denoted by line numbers, written in the form $l_1, \dots, l_i, \dots, l_n$.

Guided symbolic execution. Dynamic symbolic execution (DSE) is a program analysis technique, which intertwines symbolic execution with concrete execution. It starts with an execution path triggered by an initial test input, and then works in the loop below: from the execution path

*Corresponding author (email: ggpu@sei.ecnu.edu.cn, wkmiao@sei.ecnu.edu.cn)
The authors declare that they have no conflict of interest.

$p = l_1, \ldots, l_{i-1}, l_i, \ldots, l_n$, DSE picks an executed branch (e.g., at the statement l_i) according to the underlying path exploration strategy, and then tries to flip the original branch direction (i.e., l_i) to its opposite (denoted by $\bar{l_i}$). If the path constraint collected from $l_1, \ldots, l_{i-1}$ conjuncted with the negation of the condition at l_i is feasible, a new test input t is generated. This input can drive the program execution to follow a new path $p' = l_1, \ldots, l_{i-1}, \bar{l_i}, \ldots,$ which deviates from the original path at $\bar{l_i}$. If the target test objective is covered by p', we obtain the test input.

To counter the path explosion problem in DSE, we designed several guided path exploration strategies [7] on top of our symbolic execution engine [8]: (1) Cut-point guided search (CPGS). We identify a sequence of control points that must be passed through before reaching the target statement. These points are used as intermediate goals to narrow down the path exploration space. (2) Shortest distance branch first (SDBF). We prefer to flip the branch whose opposite branch has shortest distance toward the target statement. The intuition is that a shorter path is easier to reach the target.

Adapted model checking. Counterexample-guided abstraction refinement (CEGAR) is a model checking approach, given the program code and a property of interest, it either statically proves its correctness, or outputs a counterexample to demonstrate its violation. This CEGAR-based approach works in two phases: model checking and tests from counter-examples. It checks whether the statement l of interest is reachable such that a property q holds at l. If l is reachable, a test input can be generated from this path. Otherwise, it can conclude that no test inputs could reach l when q holds.

In our context, we treat the coverage-driven testing as path reachability checking. We instrument the property (i.e., the test obligation of a test objective) into the program, and consult the reachability from a model checker. If the test objective at the target statement l is reachable, a test input is obtained. Otherwise, it is an infeasible objective. In particular, for statement or branch coverage, the property q is set as true. For data flow coverage, a similar code instrumentation strategy [7] can also be designed.

Combining symbolic execution and model checking. In principle, DSE dynamically explores program paths to identify test inputs for feasible test objectives, but fails to cover infeasible ones and wastes testing time on them. However, CEGAR can tell the feasibility of test objectives by doing reachability checking but its performance is limited by its statically checking procedure. Thus, we combine these two state-of-the-art techniques to complement each other. In practice, our approach works as follows: (1) DSE is first used to cover as many test objectives as possible within predefined testing resources; (2) CEGAR is then used to test those remaining uncovered objectives: it can cover feasible ones as well as eliminate infeasible ones. As a result, more efficient coverage-driven testing can be achieved by further improving code coverage and reducing testing time than just using these two approaches alone.

Acknowledgements Ting SU was supported by East China Normal University Program for Outstanding Doctoral Dissertation Cultivation (Grant No. PY2015032). This work was also supported by Science and Technology Commission of Shanghai Municipality Project (Grant No. 14511100400), National Natural Science Foundation of China (Grant No. 61402178), Science and Technology Commission of Shanghai Municipality Project (Grant No. 14YF1404300), and National Major Scientific and Technological Special Project (Grant No. 2014ZX01038-101-001).

References

1 Myers G J, Sandler C. The Art of Software Testing. Hoboken: John Wiley & Sons, 2004
2 National Institute of Standards and Technology. The Economic Impacts of Inadequate Infrastructure for Software Testing. Planning Report 02-3. 2004
3 Ammann P, Offutt A J, Huang H. Coverage criteria for logical expressions. In: Proceedings of the 14th International Symposium on Software Reliability Engineering, Denver, 2003. 99–107
4 Su T, Wu K, Miao W, et al. A Survey on Data Flow Testing. Technical Report SU01. 2015
5 Godefroid P, Klarlund N, Sen K. DART: directed automated random testing. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2005. 213–223
6 Beyer D, Chlipala A J, Henzinger T A, et al. Generating tests from counterexamples. In: Proceedings of the 26th International Conference on Software Engineering, Edinburgh, 2004. 326–335
7 Su T, Fu Z, Pu G, et al. Combining symbolic execution and model checking for data flow testing. In: Proceedings of IEEE/ACM 37th International Conference on Software Engineering, Florence, 2015. 654–665
8 Su T, Pu G, Fang B, et al. Automated coverage-driven test data generation using dynamic symbolic execution. In: Proceedings of the 8th International Conference on Software Security and Reliability (SERE), San Francisco, 2014. 98–107