Casual Analysis of Residual Structural Coverage in Dynamic Symbolic Execution

ABSTRACT

High structural coverage of the code under test is often used as an indicator of the thoroughness and the confidence level of testing. Dynamic symbolic execution is a testing technique which explores feasible paths of the program under test by executing it with different generated test inputs to achieve high structural coverage. It collects the symbolic constraints along the path explored and negates one of the constraints to obtain a new path. However, due to the difficulty of method sequence generation, long run loop and testability issues, it may not be able to generate the test inputs for every feasible path. These problems could be solved by involving developers' help to assist the generation of inputs for solving the constraints. To help the developers figure out the problems, reportig every issue encountered is not enough since browsing through a long list of reported issues and picking up the most related one for the problem is not a easy task as well. In this paper, we propose an approach for carrying out the casual analysis of residual structural coverage in dynamic symbolic execution, which collects the reported issues and coverage information, filter out the unrelated ones and report the not covered branches with the associated issues. We conducted the evaluation on a set of open source projects and the result shows that our approach reported ?% related issues(may be 100% without false negative) and ?% less issues than the issues reported by Pex, an automated structural testing tool developed at Microsoft Research for .NET programs.

1. INTRODUCTION

A main objective of structural software testing is to achieve full or at least high code coverage such as statement and branch coverage of the program under test. A passing test suite that achieves high code coverage not only indicates the thoroughness of the testing but also provides high confidence of the quality of the program under test. Dynamic Symbolic Execution(DSE)[2, 4, 5] is a variation of symbolic execution, which systematically explores feasible paths of the program under test by running the program with different test inputs to achieve high structural coverage. It collects the symbolic constraints on inputs obtained from predicates in branch statements along the execution and rely on a constraint solver, Z3 for Pex[6] and STP[3] for KLEE[1], to solve the constraints and generate new test input for exploring new path. Currently, DSE works well in generating inputs for methods or parameterized unit tests with parameters of primitive type. However, when applying in object-oriented code, DSE could not easily generate inputs to achieve high structural coverage due to their little support for method sequence generation and floating point arithmetic, huge search space of feasible paths caused by loops and dependence of external library. Tackling these problems require complex analysis of the program and algorithms to find out solutions from a large possible space. But human, especially developers who write the program, could figure out the solution in a short time if provided the branch and statement coverage information with the relevant issues. Exsiting tools, like Pex, could report every issue encountered during the exploration, but some of the issues are actually not the cause of the problem. This will usually result in a long list of unordered issues, which makes it time consuming and tedious for user to figure out which action should be taken for guiding the DSE tool to increase the coverage.

In this paper, we propose an approach, Covana, which analyses the information collected during the DSE exploration, filters out the unrelevant information and report the not covered branches with the classified issues. To better inform user the problem, we define the categories of the issues which prevent the DSE technique to generate corresponding inputs:

- object creation problem due to the limitation of method sequence generation
- external library dependence, like uninstrumented method invocations
- environment dependence, like file system, database and so on
- explosion of feasible paths caused by loops or multi-level factory method (may not be examined)

Provided with the coverage information and the reported issue when DSE technique fails to generate an input for a paticular path, our approach is capable of locating the issues for a specific not covered branch and filtering out the unrelevant ones. In this way, user could browse the issues ordered by the not covered issues and target the problems directly.

Our approach is built upon Pex, an white box test input generation tool developed by Microsoft Research. In order to obtain the coverage information and the issues reported during execution, we developed a Pex extension that serves as an observer to the different events occuring along the execution, collects the information of branch coverage and reported issues and save them into binary files. Given the information in files, we developed a tool that analyses these kinds of information, classifies the issues into the defined categories and attach the related issues to the not covered branches.

Our detailed approach is:

1. Collects the branch coverage and find out the not covered branches. If the target statement of the not covered branch is covered, we simply filter out these not covered branches as it is useless for achieving higher coverage.

2. For the not covered branch which involves non-primitive object fields, we will search the reported issues to see whether there is an object creation issue for these fields. If found, we will consider this object creation issue is the cause which disallows Pex to cover the specific branch and report them with the not covered branch.

3. For the uninstrumented method, we will make the method's return value as a symbolic and keep track of it. If there are not coverd branches whose constraints involve the symbolic we tracked, we will report it as the issue of external library dependence. Otherwise we will ignore it and discard the related information.

4. When DSE tool, like pex, fails to deal with the environment dependency or create an instance for some class which implements a specific interface, our tool will suggest user to use mock objects for solving these kind of problems.

5. When there is a explosion of feasible paths caused by loops, we will suggest ? (needs more experiments)

2. EXAMPLE

3. REFERENCES

[1] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[2] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[3] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, pages 519–531, 2007.

[4] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[5] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[6] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In *TAP*, pages 134–153, 2008.