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. However, due to the difficulty of method sequence generation and testability issues, some of the constraints encountered during executions could not be easily solved, which would result in the uncoverage of the corresponding feasible paths. 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 as browsing through a 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 would classify the reported issues to different unsolved constraints, filter out the unrelated ones and rank the constraints based on the different factors, such as number of dependent blocks and number of issues related. We also provide another view of encountered issues by ranking them with the number of constraints associated. We conducted the evaluation on a set of open source projects and the result shows that our approach reported ?% 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, unsolved constraints involving external method invocations and huge search space of feasible paths. 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, may figure out the solution in a short time if provided the relevant coverage information with the issues encountered but not solved. Exsiting tools could report every issue encountered during the exploration but most of them are not directly related to the cause of the uncovered branches. This will usually result in a long list of a long list of reported 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.

To address this problem, our approach, Covana, analyses the information collected during the DSE exploration, filters out the unrelevant information and group them by each uncovered branch. We define the causes for uncovered branch into the following categories:

- object creation problem due to the limitation of method sequence generation
- external library dependence, like uninstrumented method invocations
- object creation problem for interface
- explosion of feasible paths

We will examine the collected information, split the related issues into these cause categories and filter out the unrelated ones. Our detailed action for each category is:

1. For the uncovered 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 yes, we will link the object creation issue the branch and consider it as the cause for the uncovered branch.

2. For the uninstrumented method, we will make the method's return value as a symbolic and keep track of it. If there are unsolved constraints which involves the symbolic, we will report it as the issue of external library dependence.

3. When DSE tool, like pex, fails to create an instance for some class which implements a specific interface, we will filter out the object creation issues related to this type and suggest user to use mock for solving these kind of problems.

4. When there is a explosion of feasible paths of a PUT or a PUT with factory method, we will suggest to limit the number of parameters of the PUT or the factory method.

2. 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.