Casual Analysis of Residual Structural Coverage in Dynamic Symbolic Execution

Xusheng Xiao, Tao Xie
Department of Computer Science, North Carolina State University, Raleigh, NC, 27695, USA
{xxiao2, txie}@ncsu.edu

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 non-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 coverage [1] and block coverage [7] 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) [3, 6, 8] 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 [9] and STP [5] for KLEE [2], 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.

To address this problem, we propose an approach, Covana, which analyses the data collected during the DSE exploration, filters out the unrelevant data and report the non-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 data 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 to build an analysis tool, Covana, upon Pex, an white box test input generation tool developed by Microsoft Research. In order to obtain the coverage data and reported issues from Pex, we provides several Pex extensions that server as observers for observing events and collecting data. Based on the collected data, our approach will pre-processed the data and carry out the analysis on the pre-processed data.

2. EXAMPLE

Will fill in later...

3. APPROACH

3.1 Approach Overview

Our approach takes the coverage data and reported issues as input and outputs the non-covered branches with the associated issues. Figure 1 shows the high level design of Covana. Covana consists of three major components: observer component, pre-processing component and analysis component.The observer component is implemented as several Pex extensions and is attached to Pex for observing different events and collecting data. It collects the coverage data and reported issues and passed them to the pre-processing component. The pre-processing component remove the useless data and dump the data into binary files. The analysis component consumes the pre-processed data in files, carrys out the analysis on the data and outputs the relevant data for non-covered issues.

3.2 Observer Component

The observer component is implemented as three Pex extensions: (1) Branch coverage observer. (2) Field access observer. (3) Issues observer.

3.2.1 Branch coverage observer

The branch coverage observer is attached to Pex for collecting the branch coverage data. After all the generated tests are executed, it will enumerate the methods of the class under test, checks the coverage information of each branch inside a method and find out the non-covered branches. As Pex operates on the level of MSIL (MicroSoft Intermediate Language) [4], the observer also collects the IL offset of each non-covered branches for further analysis.

3.2.2 Field access observer

During the execution with the initial simple input, Pex collects the symbolic constraints along the path explored and flips one of the constraints to obtain a new path for the future executions. The field access observer is pluged into Pex for observing the events raised when Pex could not flip a constraint to obtain a new path. It will collect the information about the branch where Pex fails and the involved fields.

3.2.3 Issues observer

The issue observer collects different kinds of issues reported by Pex, such as object creation issues, uninstrumented method issues and testability issues.

```
if (x != null) {
  Console.WriteLine("type: " + x.GetType());
}
IEnumerator xe = x.GetEnumerator();
```

Figure 1: Non-covered branch with full block coverage when x is not null

```
if (x == null || y == null) {
  return true;
}
return false;
```

Figure 2: Non-covered branch with full block coverage when x is not null but y could be null

3.3 Pre-processing Component

The pre-processing component pre-processes the different kinds of data collected by the observer component, filters out the useless data based on some heuristics and save the data into binary files.

3.3.1 Filter out useless branch coverage data

A test suite satisfying all branch coverage criteria always satisfies all statement coverage criteria, but not vice versa. Thus, when we get the data of all the non-covered branches, not all of them are useful for user since we focus on helping user to achieve higher block or statement coverage.

Figure 1 shows an example of code under test which achieves full block coverage but not branch coverage. In the example, x is the class under test and is assumed to be not null. In this case, if x is not null, the test case achieves full block coverage but not full branch coverage since the false branch of "x != null" is not covered. However, even we make this branch covered, it won't increase the statement or block coverage as it is already covered. Hence, we could simply filter out such non-covered branches. To find out such kind of non-covered branches, we need to check the coverage of their target statements. If their target statements are already covered, then these non-covered branches are considered useless and could be safely filtered out. (later we may deal with the situation of multi clause, like "if(x!=null || y != null)" showed in Figure 2)

3.3.2 Filter out field access information of covered branches

The field access information is collected when Pex fails to flip a constraint on a branch. However, Pex does not just stop exploration due to its failing to flip a constraint. It will try to flip other constraints and this may result in a path which covers the branch. Hence, when the field access information going through the pre-processing component, we will filter out the filed access information of the covered branches.

3.4 Analysis Component

The analysis component reads the pre-processed data from binary files and carrys out the analysis on it. It filters out the unrelevant issues, classifies issues into the defined categories and reports the non-covered issues with the related issues.

3.4.1 Object Creation Issue

To find out whether an object creation issue is related to a non-covered branch or not, we need to examine the information about the field access information for the non-covered branches and object creation issues reported. As the preprocessed component has filtered out the field access information of the covered branches, the remaining field access information could be used directly for the analysis. Our approach thus picks up object creation issues one by one and check whether the type of each reported object is the same as the type of some field involved in non-covered branches. If yes, we consider the this object creation issue is relevant to the non-covered branch and assign it to the branch, which will be reported together later. Otherwise, we just simply ignore the issue.

3.4.2 External Library Dependency Issue

We use the latest feature of Pex to track the return value of each uninstrumented method. If it is involved in some branch where Pex fails to cover, then we will report it. (This part could be implemented using Nikolai's newly released feature of Pex)

3.4.3 Environment Dependency Issue

We could check whether the non-covered branch involves some static method call(File.Exists) or method calls of some environment dependency objects (socket for network connection, ADO.NET for database and other common .NET library for environment interaction). We could define a set of these library names and if we find them in some non-covered branch, we could simply consider this is the environment dependency issue.

3.4.4 Loop Issue

This may not be so easy. But the current clue I find is that inside a long run loop, the problem information collected when Pex tries to solve a problem will have many similar constraints. The code I use for testing is showed in Figure 3. Figure 4 shows the problem log I got. They all occurs at the same line and the reason why they always choose to flip the n.length is because of the fitness strategy. To know whether there is a loop issue or not, we could check whether Pex report path boundary or not.

```
public static void LargeLoop(int[] number)
{
     for (int i = 0; i < 5000; i++)
     {
          for (int j = 0; j < number.Length - i -
1; j++)
          {
               if (number[j] > number[j + 1])
               {
                    int temp = number[j];
                    number[j] = number[j + 1];
                    number[j + 1] = temp;
               }
          }
     }
     if (number.Length > 5000)
     {
          throw new Exception("bug");
     }
}
```

Figure 3: Long run loop for test

```
flipped location:  line: 12
return 59 < n.Length;

flipped location:  line: 12
return 60 < n.Length;

flipped location:   line: 12
return 61 < n.Length;

flipped location:  line: 12
return 62 < n.Length;

flipped location:  line: 12
return 63 < n.Length;

flipped location:  line: 12
return 64 < n.Length;
```

Figure 4: Part of the problem log of Pex for long run loop

4. REFERENCES

[1] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[2] 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.

[3] 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.

[4] Jonathan de Halleux and Nikolai Tillmann. Parameterized unit testing with pex. In *TAP*, pages 171–181, 2008.

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

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

[7] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, 14(6):868–874, 1988.

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

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