Casual Analysis of Residual Structural Coverage

ABSTRACT

Dynamic symbolic execution is a testing technique which exercises different execution paths of the program under test by executing it with generated test inputs. By writing parameterized unit tests, which takes inputs and states assumption and assertions, a tool using dynamic symbolic execution could create a test suite with high code coverage by genrating test inputs to satisfy different path conditions. However, solving some of path conditions require desirable method sequences to create and modify objects, which is often challenging due to the large search space of possible sequences. The other path conditions may not be easily satisfied due to testability issues as well, such as solving constraints which depends on the result of method from external library or running out of time before exit the loop. Failing to solve these issues would result in the uncoverage of the corresponding feasible paths. In this paper, we propose an approach for carrying out the Casual Analysis of Residual Structural Coverage in Dynamic Symbolic Execution, which could provide the information of the causal effect chain for the specific uncovered branch and help developers to make it covered. For an uncovered branch, the reason why it could not be covered by adopting dynamic symbolic execution could be just one testability problem or the combination of them, like requiring method sequences which involves methods from external library as well. Based on the information collected by running the tool which uses dynamic symbolic execution, our approach could narrow down the issues reported and point out which one are relevant for helping achieve residual structural coverage. Using this information, developers could assist the tools to cover the branch much easier. Implementation and experiment evaluation comes here ...

1. INTRODUCTION

To explore the feasible paths of the program under test, dynamic execution tool collects the constraints encountered during the execution and rely on a constraint solver, Z3 for pex and STP for klee, to solve the constraints and generate new test input to explore new path and achieve high coverage. However, when applying in real applications, existing tools coule not cover some branches due to their little support for generating method sequences to create and modify objects, little support for floating point arithmetic or constraints involving external method invocations. Solving these problems require complex algorithms to find out the solution from a large possible space but human, especially developers who write the program, may figure the solution out in a short time if provided the relevant coverage information. Exsiting tools reports every issue encountered during the exploration and most of them are not so relevant. By browsing through a long list of reported issues, it is difficult to figure out which action should be taken to increase the coverage.

To address this problem, our approach, Covana, analyses the information collected during the execution, filters out the unrelevant information and list them in well structured format.

Description of the approach here.....

The objective of our approach is that if provided all the coverage information, like pex, developers need several changes and rerun the tool several times to make the branch covered, then adopting our approach could make the developers do it in fewer times or only one time.

Here is my initial idea:

1. Suggest the object based on the analysis of creation dependency.

```
class Parent{ private Child child;

public CreateChild(){ child = new Child(); }

public SetChildValue(int value){ child.value = value; }

}
```

For this code snippet, we could see that the creation and modification of child object totally depends on its parent. Thus, if encountering some constraints involving child object's value, we should suggest the object creation of the parent but not the child. If the parent class provides constructor injection or setter injection for child object, then we should suggest improve the creation of the child object.

This scenario also could be seen when constructor of child object is not visible.

Solving this analysis problem could be a subtask.

2. When applying pex in NHibernate, I found that when pex suggest creating an object with multiple fields, it will list all the fields in that class. But actually most of the fields are not relevant to the uncovered branch and could be filtered out. We could isolate these unrelevant fields and just list the fields involved in the constraints of the uncovered branch.

3. When applying pex in NHibernate, I found that inside a method under test, there are several other method invocations, which increases the search space a lot for pex. However, some of the methods are not from the object under test and pex still tries to cover all the branches inside such methods.

Supposed we have a class parent with an uncovered branch which requires the evaluation of the child's method, to satisfy this constraint we only need to make the child's method return true.

```
public class Parent
{
    private Child child;

    public Parent(Child child)
    {
        this.child = child;
    }


    public string SaySomething(string value)
    {
        if (child.CanUnderstand(value))
        {
            return "good " + value;
        }
        return "bad " + value;
    }
}
```

```
public bool CanUnderstand(string value)
{
    if (value.Contains("hi"))
    {
        return true;
    }

    if (value.Contains("a"))
    {
        if (value.EndsWith("b"))
        {
            return true;
        }
        return false;
    }

    if (value.Contains("b"))
    {
        if (value.EndsWith("a"))
        {
            return true;
        }
    }
    return false;
}
```

After digging into child's method, we could see that the method could have several paths to return true. I think here we do not need to explore every possible path of class child as our test target is parent, especially when child's methos is an uninstrumented method. We could just choose two values which statisfy the predict coverage of the child's method and the coverage of the parent would be fine. Currently pex will report every issue encountered during the exploration, no matter it is the class under test or not. In this simple example, pex is able to find out all the possible inputs for the child's method, CanUnderstand. However, in real application, like NHibernate or Quartz.NET, the call stack of external method invocation is deep and it is often involved many object creation issues, which make it impossible for pex to solve the constraint. Therefore, if we only satisfy the predict coverage of external method, like child's method, then we could reduce the pex's effort a lot.

To provide more relevant information for developers, we could do static analysis of the external method. Taking the child's method, CanUnderstand, in figure 1 for example, to satisfy predict coverage, we only need to find out two path which make it return true and false respectively. Supposed pex is stuck for two constraints inside the external method, then we should rank the constraints and give the one which have fewer condition checks before it reaches the end of the method higher score. In the given example, supposed we want the method to return true and pex could not solve all

the constraints that make it return true, we should rank the first constraint, value.Constains(“hi”), as there is no other condition check following it.