Problem Identification for Structural Test Generation: First Step Towards Cooperative Developer Testing

Xusheng Xiao
xxiao2@ncsu.edu
Department of Computer Science
North Carolina State University, Raleigh, NC, USA

ABSTRACT

Achieving high structural coverage is an important goal of software testing. Instead of manually producing high-covering test inputs that achieve high structural coverage, testers or developers can employ tools built based on automated test-generation approaches to automatically generate such test inputs. Although these tools can easily generate high-covering test inputs for simple programs, when applied on complex programs in practice, these tools face various problems, such as the problems of dealing with method calls to external libraries, generating method-call sequences to produce desired object states, and exceeding defined boundaries of resources due to loops. Since these tools currently are not powerful enough to deal with these various problems in testing complex programs, we propose cooperative developer testing, where developers provide guidance to help tools achieve higher structural coverage. To reduce the efforts of developers in providing guidance to the tools, we propose a novel approach, called Covana. Covana precisely identifies and reports problems that prevent the tools from achieving high structural coverage primarily by determining whether branch statements containing not-covered branches have data dependencies on problem candidates.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Measurement, Reliability

Keywords

Structural test generation, dynamic symbolic execution, data dependency, problem identification

1. RESEARCH PROBLEM

Achieving high structural coverage is an important goal of software testing. Manually producing such test inputs is labor-intensive. To address the issue, tools built based on automated test-generation approaches can be employed to automatically generate test inputs, such as Dynamic Symbolic Execution (DSE) [5, 6, 10] (also called concolic testing [10]) and random approach [3, 7]. Although these tools can easily achieve high structural coverage for simple programs, these tools face challenges in generating high-covering test inputs to achieve high structural coverage when applied on complex programs in practice. Our preliminary study [15] shows that many statements or branches are not covered due to two major types of problems: (1) external-method-call problems (EMCP), where tools cannot deal with method calls to external libraries to achieve high coverage; (2) object-creation problems (OCP), where tools fail to generate sequences of method calls to construct desired object states to cover certain branches. Our study also identifies boundary problems (BP) as another important type of problems, where tools exceed pre-defined boundaries on resources before achieving high coverage, often caused by loops.

Since these tools are imperfect in dealing with various problems in complex programs without human intervention, we propose a new methodology of cooperative developer testing, where developers provide guidance to help the tools address the problems. For example, to deal with BPs, developers can increase the pre-defined boundaries of resources or limit the number of iterations that a loop should take. To acquire developers' guidance, the tools need to report the encountered problems. For example, the tools can report all the external-method calls in the program under test. However, the number of such problem candidates could be high, and some of these problem candidates are not causes for the tools not to achieve high structural coverage; these candidates are referred to as irrelevant problem candidates. As an example, the external-method call `Console.WriteLine` only prints the string value of the argument.

To address the need of reducing the efforts of developers in providing guidance to tools, we propose a novel approach, called Covana [15]. Covana precisely identifies problems that prevent the tools from achieving high structural coverage and prunes irrelevant problem candidates using data dependencies of branch statements containing not-covered branches (referred to as partially-covered branch statements). Covana consists of three main steps: (1) identify problem candidates based on the types of problems, (2) assign symbolic values to elements of the problem candidates (such as return values of external-method calls) and perform forward symbolic execution [12] using the generated test inputs of the tools, (3) compute data dependencies of partially-covered branch statements on problem candidates and prune problem candidates (called irrelevant ones) that partially-covered branch statements have no data dependencies on.

2. APPROACH

We use Dynamic Symbolic Execution (DSE) [1,4–6,10,12] as an illustrative example of automated test-generation approaches, due to recent growing research on DSE. DSE instruments and executes the program under test symbolically, collecting constraints on program inputs to form path conditions and negating part of the collected path conditions to obtain new paths for further exploration. We concretize Covana as an extensible framework that collects information from DSE to identify different types of problem candidates, assigns symbolic values to elements of the identified problem candidates, such as return values of external-method calls, and collects the constraints on these symbolic values from subsequently executed branches to determine whether partially-covered branch statements have data dependencies on problem candidates for their elements. We further propose three techniques that can be fed into Covana for identifying EMCPs, OCPs and BPs.

2.1 EMCP Identification

Problem Candidate Identification. We consider as problem candidates only external-method calls whose arguments have data dependencies for program inputs. Normally, external-method calls whose arguments have no data dependencies for program inputs are method calls that print constant string or put a thread to sleep for some time. These method calls do not prevent DSE from achieving high structural coverage and can be safely pruned.

Data Dependence Analysis. In our preliminary study, we observe that if the return values of external-method calls are used to decide certain branches, these branches are very likely not covered by the generated test inputs, since automated test-generation tools normally cannot explore called external methods without instrumenting them. Hence, Covana assigns symbolic values to the return values of the identified external-method calls for computing data dependencies for the return values. Covana prunes EMCP candidates if none of partially-covered branch statements have data dependencies on the candidates for their return values.

Uncaught Exception Analysis. Uncaught exceptions thrown by external-method calls abort test executions, preventing DSE from exploring remaining parts of the program under test. To identify such external-method calls, Covana collects exceptions during test execution and analyzes the stack trace to extract external-method calls. If the remaining parts of the program after the call sites of the external-method calls are not covered, Covana reports these external-method calls as EMCPs.

2.2 OCP Identification

Problem Candidate Identification. Since OCP requires objects of a non-primitive type as program inputs, we consider as problem candidates only program inputs whose type is a non-primitive type, such as a user-defined type. Covana assigns symbolic values to such program inputs and all their fields for computing data dependencies. In addition, Covana collects path conditions during the test execution for further analysis.

Data Dependence Analysis. In our preliminary study, we observe that some branches may require only a specific object state of only a field of the program inputs, and thus there is no need to spend effort in providing sequences of method calls for all the fields of program inputs. By computing data dependencies of partially-covered branch statements for the fields of program inputs, we identify some fields of the program inputs as the OCP candidates.

```
00:for(int i = 0; i < input;i++) { ... }
01:if(input > 1000) { // simplified away after loop
02:  // not-covered area }
```

Figure 1: Example of Boundary Problem (BP)

Path Condition Analysis. To identify which field requires guidance to provide desired object states, Covana further analyzes the colleced path conditions that lead to not-covered branches. From the path conditions, Covana extracts the fields of program inputs and constructs a field declaration hierarchy up to the program input. If a field cannot be assigned with an object directly by invoking a constructor or a public setter method of its declaring class, the object state of the field can be changed only by invoking other public state-modifying methods of its declaring class. Hence, Covana reports its declaring class type as an OCP. Otherwise, Covana continues to check the next field in the hierarchy.

2.3 BP Identification

Motivating Example. In our preliminary study, we observe that if there exist some loops whose number of iterations has data dependencies for program inputs, DSE keeps negating the constraint to increase the number of iterations for the loops. As a side effect, the constraints on program inputs in subsequent executed branches are simplified away (as illustrated with an example below), preventing DSE from analyzing the constraints for exploring remaining parts of the program.

Figure 1 shows a simplified example of BP. Assuming in an execution, the concrete value of `input` is N. When the loop terminates, the path condition must contain two constraints: (1) $N - 1 < input$ for the last loop iteration; (2) $N >= input$ for the loop termination. Combining these two constraints implies $N == input$, which is in turn used to simplify $input > 1000$ to either true or false. To cover the not-covered area, DSE has to negate the constraint for extending the loop until $input > 1000$ is satisfied, which probably cannot be achieved before DSE exceeds the pre-defined boundary on the number of exploration paths.

Problem Identification. To identify such situations, every time DSE selects a particular branch and negates its constraint, Covana considers the variables involved in the constraint as problem candidates. In the example shown in Figure 1, `input` is identified as the problem candidate. If later in this exploration, DSE exceeds a pre-defined boundary on resources, and some partially-covered branch statements have data dependencies for the variable `input`, Covana considers the problem candidate as a BP.

3. RELATED WORK

Pavlopoulou and Young [8] developed a residual coverage monitoring tool for Java, which provides richer feedback from deployed software and aims to reduce the performance overhead for gathering structural coverage from deployed software. Although their approach analyzes residual structural coverage, their approach does not provide a way to analyze the coverage, while our approach analyzes the residual structural coverage gathered from test-generation tools to filter out irrelevant problem candidates. Dincklage and Diwan [14] propose an analysis language and build a system to produce reasons when the program analysis fails to produce desirable results. Although our approach is remotely related to their approach in terms of helping explain causes of residual structural coverage in the form of problems, our approach focuses on a significantly different problem and proposes significantly different techniques for addressing unique challenges in identifying problems for structural test generation. Anand et al. [2] propose an approach that identifies problematic external-method calls in symbolic execution by carrying out static analysis to determine whether an external-method call receives symbolic values as arguments. To identify EMCPs, our approach considers not only the data dependencies of arguments of external-method calls for program inputs, but also the data dependencies on external-method calls for their return values.

4. RESULTS AND CONTRIBUTIONS

Our work is the first to provide advice to developers by using dynamic analysis results for automated test-generation tools. We implemented the two techniques of identifying EMCPs and OCPs [15], and conducted evaluations on xUnit [16] and QuickGraph [9]. Our results show that Covana effectively identifies 43 EMCPs out of 1610 EMCP candidates with only 1 false positive and 2 false negatives, and 155 OCPs out of 451 OCP candidates with 20 false positives and 30 false negatives.

Besides identifying problems for DSE, the output of our approach can also assist other test-generation approaches. The first example is to assist automatic mock object generation. Since Covana greatly reduces the number of irrelevant problem candidates of EMCP, thus reducing the candidate space, it becomes feasible for generating mock objects [13] for all the external-method calls identified as EMCPs. As another example, a random testing approach can assign more probabilities on exploring the object types reported by Covana as OCPs, increasing the chances to achieve higher structural coverage in shorter time. Advanced method-sequence-generation approaches [11] can also be used to address OCPs to increase coverage.

In our evaluations of identifying EMCPs and OCPs, we identify two issues that affect the effectiveness of our approach: (1) **static fields**: the static fields are initialized inside the declaring class and may be later used by some branches. Some of these branches are not covered because the value of a static field is changed by the tests executed before the current test; (2) **concrete argument for an external-method call**: some of such external-method calls use constant values to access external environment states and cause some branches not to be covered.

Although we do not encounter other issues in our evaluations, there are still some potential issues that may affect the effectiveness of our approach: (1) **argument side effect**: some external-method calls may have side effects on the receiver objects or method arguments have data dependencies on program inputs, causing some subsequent branches not to be covered; (2) **control dependency**: extending our approach to consider control dependency may improve the effectiveness of our approach in some cases; (3) **static analysis**: our approach currently computes dynamic data dependencies based on the executed paths, which may miss some data dependencies on unexecuted paths. Employing static analysis to analyze all the paths is one option to solve the problem. Nevertheless, due to the complexity of programs, static analysis may produce false positives on detecting data dependencies, which in turn affect the effectiveness of our approach. We plan to conduct experiments to evaluate the effectiveness of incorporating static analysis.

We have started working on the implementation of the third technique to identify BPs and plan to evaluate this new technique. We are also working on a novel visualization approach to better assist developers in locating identified problems and providing guidances.

5. REFERENCES

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven Compositional Symbolic Execution. In *Proc. TACAS*, pages 367–381, 2008.

[2] S. Anand, A. Orso, and M. J. Harrold. Type-Dependence Analysis and Program Transformation for Symbolic Execution. In *Proc. TACAS*, pages 117–133, 2007.

[3] C. Csallner and Y. Smaragdakis. JCrasher: An Automatic Robustness Tester for Java. *Software—Practice & Experience*, pages 1025–1050, 2004.

[4] P. Godefroid. Compositional Dynamic Test Generation. In *Proc. POPL*, pages 47–54, 2007.

[5] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proc. PLDI*, pages 213–223, 2005.

[6] J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.

[7] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *Proc. ICSE*, pages 75–84, 2007.

[8] C. Pavlopoulou and M. Young. Residual Test Coverage Monitoring. In *Proc. ICSE*, pages 277–284, 1999.

[9] QuickGraph, 2008. http://www.codeproject.com/KB /miscctrl/quickgraph.aspx.

[10] K. Sen, D. Marinov, and G. Agha. CUTE: a Concolic Unit Testing Engine for C. In *Proc. ESEC/FSE*, pages 263–272, 2005.

[11] S. Thummalapenta, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. MSeqGen: Object-Oriented Unit-Test Generation via Mining Source Code. In *Proc. ESEC/FSE*, August 2009.

[12] N. Tillmann and J. de Halleux. Pex-White Box Test Generation for .NET. In *Proc. TAP*, pages 134–153, 2008.

[13] N. Tillmann and W. Schulte. Mock-object Generation with Behavior. In *Proc. ASE*, pages 365–368, 2006.

[14] D. von Dincklage and A. Diwan. Explaining Failures of Program Analyses. In *Proc. PLDI*, pages 260–269, 2008.

[15] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise Identification of Problems for Structural Test Generation. In *Proc. ICSE*, 2011.

[16] xUnit, 2007. `http://www.codeplex.com/xunit`.