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. To ease the task of manually producing high-covering test inputs that achieve high structural coverage, tools built based on automated test-generation approaches can be employed to automatically generate such test inputs. Although such 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: 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, 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.

1. PROBLEM AND MOTIVATION

Achieving high structural coverage is an important goal of software testing. Manually producing such test inputs is labor-intensive. To address the issue, tools based on automated test-generation approaches, such as Dynamic Symbolic Execution (DSE) [8, 10, 18] (also called concolic testing [18]) and random approaches [5,13,14], can be employed to automatically generate test inputs.

Although such tools can easily achieve high structural coverage for simple programs, they face challenges in generating high-covering test inputs when applied on complex programs in practice. Our preliminary study [26] indicates 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 for achieving 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 further 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 OCPs, developers can specify factory methods [22] that encode method sequences for producing desired object states. To deal with EMCPs, developers can instruct the tools to instrument the external-method calls or provide mock objects [24] to simulate irrelevant environment dependencies. To deal with BPs, developers can increase the pre-defined boundaries of resources or limit the number of iterations that a loop should take by specifying assumptions. After providing guidance to the tools based on the reported problems, developers can reapply tools to generate test inputs for achieving better coverage. Such iterations of applying tools and providing guidance can continue until desired coverage is achieved.

To acquire developers' guidance, the tools need to report the encountered problems. For example, the tools can report all the external-method calls or all non-primitive object types in the program under test. However, the number of such problem candidates could be large, and some of these problem candidates may not prevent the tools from achieving high structural coverage. These candidates are hereby referred to as irrelevant problem candidates. For example, DSE cannot explore the external-method call `Console.WriteLine` of C# projects. However, instructing DSE to explore `Console.WriteLine` does not increase the structural coverage, since `Console.WriteLine` only prints the string value of the input argument and does not affect the coverage in subsequent branches. As another example, certain branches may require only the specific object state of a field of a program input. Thus, there is no need to spend effort in generating object states for other irrelevant fields of program inputs.

To reduce the efforts of developers in providing guidance to tools, we propose a novel approach, called Covana [26]. Covana precisely identifies problems that prevent structural test-generation tools from achieving high structural coverage and prunes irrelevant problem candidates using data dependencies of branch statements containing not-covered

Project	LOC	Cov %	OCP	EMCP	BP	Limitation
SvnBridge	17.1K	56.26	11 (42.31%)	15 (57.69%)	0 (0%)	0 (0%)
xUnit	11.4K	15.54	8 (72.73%)	3 (27.27%)	0 (0%)	0 (0%)
Math.Net	3.5K	62.84	17 (70.83%)	1 (4.17%)	4 (16.67%)	2 (8.33%)
QuickGraph	8.3K	53.21	10 (100%)	0 (0%)	0 (0%)	0 (0%)
Total	40.3K	49.87	46 (64.79%)	19 (26.76%)	4 (5.63%)	2 (2.82%)

Table 1: Main problems for not-covered branches in 10 files from core libraries of four open source projects

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 [22] 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. BACKGROUND AND RELATED WORK

I now describe two state-of-the-art structural test-generation approaches (random testing [5, 13, 14] and DSE) and a preliminary study of applying DSE on open source projects, and then compare my approach to other related work.

2.1 Structural Test-Generation Approaches

Structural test-generation approaches aim to produce test inputs to achieve high structural coverage of the program under test, such as statement or branch coverage [1]. Random testing [5,13,14] randomly selects test inputs in a given range. Random testing is scalable and able to find previously identified faults automatically [13, 14]. However, random testing usually achieves lower coverage compared to a systemic test-generation approach, such as DSE [8, 10, 18]. The main reason is that certain branches in the program under test may have a low probability of being covered by randomly generated test inputs.

DSE executes the program under test symbolically with arbitrary or default inputs. Along the execution path, DSE collects the constraints in the branch statements to form a path condition and negates part of the path condition to obtain a new path condition. The new path condition is then fed into a constraint solver, such as Z3 [6] or CVC3 [4], which computes new test inputs to steer future executions toward new paths. Ideally, all feasible paths can be exercised eventually through such iterations of path variations.

2.2 Preliminary Study

We carried out a preliminary study for assessing the performance of structural test-generation tools on complex programs. In particular, we applied Pex [22], a state-of-the-art DSE tool, on four open source projects, and observed different types of problems faced by Pex. We chose Pex as the structural test-generation tool for the empirical study due to following reasons: (1) DSE achieves better code coverage compared to random approaches; (2) Pex has been applied internally in Microsoft to test core components of the .NET runtime infrastructure and found serious defects [22].

We applied Pex to generate test inputs for the core libraries of the subject projects until either all the methods had been explored by Pex, or Pex ran out of memory and could not continue to generate test inputs. Our subject projects included SvnBridge [19], xUnit [27], Math.NET [12], and QuickGraph [16]. These subject projects are quite popular, as indicated by high download counts. The details of the subject projects and results can be found in our project website[1].

Table 1 shows the results of the preliminary study. Column "Project" lists the name of each project, Column "LOC" shows the number of lines of codes for each project, and Column "Cov %" shows the block coverage achieved by Pex. The other four columns give the number and the percentage of the not-covered branches caused by different types of problems. The results show that the total block coverage achieved is 49.87%, with the lowest coverage being 15.54%.

The top major type of problems is OCPs (64.79%), shown in Column "OCP". In unit testing of object-oriented code, achieving high structural coverage requires desired object states to cover various branches. However, automated approaches are often ineffective in generating method-call sequences that produce desired object states to achieve high structural coverage [21], resulting in OCPs.

The second major type of problems is EMCPs (26.76%), shown in Column "EMCP". External-method calls cause DSE to lose track of computed symbolic values passed as their arguments (resulting in inaccurate symbolic analyses), or throw exceptions to hinder the exploration. In the study, we encountered many external-method calls, 405 in 40 files, but only 4.7% (19 in 405) are causes for DSE not to achieve high structural coverage. If my approach simply reports every encountered external-method call as an EMCP, it can produce many irrelevant problems for developers to inspect.

The third main type of problems is BPs (5.63%), shown in Column "BP". BPs are mostly caused by loops in the program under test. Some programs under test have loops whose number of iterations depends on symbolic values, and DSE keeps increasing the number of iterations of the loops during path exploration, preventing DSE from exploring other paths in the remaining parts of the program.

The last main type of problems is limitations of the used constraint solver (2.82%), shown in Column "Limitation", since the used constraint solver cannot compute exact solutions to floating-point arithmetics or high-order expressions.

2.3 Related Work

Pavlopoulou and Young [15] develop 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 my approach analyzes the residual structural coverage gathered from test-generation tools to filter out irrelevant problem candidates.

Dincklage and Diwan [25] propose an analysis language and build a system to produce explanations when the program analysis fails to achieve desirable results. Although my

[1]http://research.csc.ncsu.edu/ase/projects/covana/

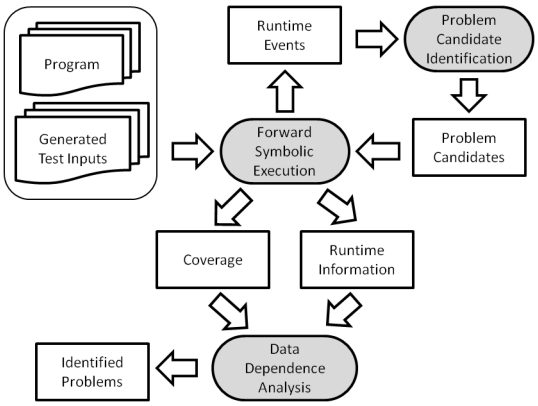

Figure 1: Overview of Covana

approach is remotely related to their approach in terms of helping explain causes of residual structural coverage in the form of problems, my approach focuses on a significantly different problem and includes significantly different techniques for addressing unique challenges in identifying problems for structural test generation.

Anand et al. [3] 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, my approach considers not only the data dependencies of external-method calls' arguments on program inputs, but also the data dependencies of partially-covered branch statements on external-method calls for their return values.

Saxena et al. [17] propose an approach that introduces symbolic variables for the iteration counts of each loop, and links these iteration counts with features of a user-provided input grammar such as variable-length or repeating fields. Godefroid et al. [9] provide a loop summarization approach that summarizes a loop as a precondition that defines a set of executions covered by the summary, and a postcondition capturing partially side-effects occurring during those executions (i.e., the effects on some variables after loop executions). While their approaches focus on directly improving symbolic execution, our approach identifies BPs when loops cause DSE not to cover subsequent branches. Thus, their approaches can be used to automatically solve some BPs, reducing the potential number of BPs identified by our approach.

3. APPROACH AND UNIQUENESS

In this section, I describe how Covana identifies problem candidates for structural test-generation tools and prunes irrelevant problem candidates using data dependencies. I next describe the overview of Covana and the analysis techniques for identifying EMCPs, OCPs and BPs.

3.1 Overview of Covana

In this work, I use DSE as an illustrative example of structural test-generation approach, because recent research [2, 7–10, 18, 22] shows that DSE generally achieve higher coverage than other test-generation approaches, such as random approaches. Figure 1 shows a high-level overview of Covana, which is concretized as an extensible framework for identifying problems faced by DSE-based tools. Covana consists of three main steps: Problem-Candidate Identification, Forward Symbolic Execution, and Data Dependence Analysis. Covana accepts as input a program under test, and generated test inputs from automated test-generation tools (such as a DSE-based tool). Covana then uses the generated-test inputs as program inputs, and leverages the DSE engine to perform forward symbolic execution on the program (program inputs are assigned with symbolic values). During execution, Covana monitors runtime events triggered by the DSE engine for identifying different types of problem candidates. After identifying problem candidates, Covana assigns symbolic values to elements of these problem candidates (such as return values of external-method calls and fields of program inputs), performs forward symbolic execution on these symbolic values, and collects runtime information, such as symbolic expressions and exceptions. Covana then uses the collected structural coverage and runtime information to compute the data dependencies of partially-covered branch statements on problem candidates. The computed data dependencies are then used to prune irrelevant problem candidates.

3.2 EMCP Identification

Problem Candidate Identification. An external method call is considered as a problem candidate if its arguments have data dependencies on 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 structural test-generation tools from achieving high structural coverage and can be safely pruned.

Data Dependence Analysis. In the preliminary study, I observed that, if the return value of an external-method call is used to decide certain branches, such branches are very likely not covered by the generated test inputs, since automated test-generation tools normally cannot explore external methods without instrumenting them. Hence, Covana assigns symbolic values to the return values of the identified external-method calls for data dependence analysis. 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 structural test-generation tools from exploring remaining parts of the program under test. To identify such external-method calls, Covana collects exceptions during test executions and analyzes the stack traces 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.

3.3 OCP Identification

Problem Candidate Identification. Since OCP requires objects of a non-primitive type as program inputs, Covana considers 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.

Data Dependence Analysis. In the preliminary study, I observed that some branches may require 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. If a partially-

```
00:for(int i = 0; i < input;i++) {  // input is symbolic
01:    c = c + 1; }
02:if(input > 1000) { // simplified away after loop
03:  // not-covered area }
04:if(c > 5000) { // not symbolic
05:  // not-covered area }
```

Figure 2: Example of Boundary Problems (BPs).

covered branch statement has data dependencies on only program inputs, Covana directly reports the program inputs as OCPs. However, if a partially-covered branch statement has data dependencies on a field of a program input, Covana performs further analysis to identify which field of the program input is the cause to the OCP.

OCP Analysis. To identify which field of the program input causes the OCP, Covana first constructs a field declaration hierarchy of the field that some partially-covered branch statements have data dependencies on. Given a field f and the class structure of a program input p, a field declaration hierarchy is formed by the fields along the path from the class type of a program input p to f. Covana then analyzes the field declaration hierarchy level by level, starting from the field of 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.

3.4 BP Identification

Motivating Example. In the preliminary study, I observed that if there exist some loops whose number of iterations has data dependencies on program inputs (referred to as input-dependent loops), 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 2 shows a simplified code snippet containing BPs. Assuming in an execution, the concrete value of *input* is N ($N > 1$). When the loop terminates, the path condition must contain two constraints: (1) $input > N - 1$ for the last loop iteration; (2) $input <= N$ for the loop termination. Combining these two constraints implies $input == N$, which is in turn used to simplify $input > 1000$ to either true or false. To cover the not-covered area at Line 3, 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.

In Figure 2, another example of BP is caused by the variable c at Line 3. The value of c is dependent on the value of *input*, since c is increased by 1 in each iteration of the loop. However, since c has no direct data dependences on *input*, c is not assigned with a symbolic value of *input*, and $c > 5000$ is not collected as a symbolic constraint for analysis. To cover the not-covered area at Line 5, DSE has to negate the constraint for extending the loop until $c > 5000$ is satisfied, which results in a BP similar to the example of satisfying $input > 1000$.

Problem Identification. To identify BPs, Covana employs pattern-matching rules introduced in the loop summary approach [9] to first identify input-dependent loops and express their iteration counts as symbolic expressions. For example, the iteration count of the loop in Figure 2 can be expressed as the value of *input*. Covana then computes side-effects occurring during the loop executions, i.e., which variables can be expressed using the iteration counts, capturing c in the example. 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.

4. RESULTS AND CONTRIBUTIONS

In this section, I describe the major contributions and results of my work, and discuss about how my work contributes to other computer science research and directions for future work.

4.1 Major Contributions

To the best of our knowledge, our work is the first attempt to precisely identify problems faced by structural-test-generation tools, achieving the first step towards cooperative developer testing between tools and developers. We provide a novel approach, called Covana, which identifies different types of problem candidates and prunes irrelevant candidates by computing data dependencies using forward symbolic execution. We also concretizes Covana as a framework to identify problems that prevent DSE-based tools from achieving high structural coverage, and provide analysis techniques to identify EMCPs, OCPs, and BPs.

4.2 Results

I implemented the proposed techniques of identifying EMCPs and OCPs as an extension to Pex [26], and conducted evaluations on xUnit [27] and QuickGraph [16]. The results show that Covana effectively identifies 43 EMCPs out of 1610 EMCP candidates, and 155 OCPs out of 451 OCP candidates. The results also show that Covana effectively prunes 97.33% (1567 in 1610) EMCP candidates with only 1 false positive and 2 false negatives and prunes 65.63% (296 in 451) OCP candidates with 20 false positives and 30 false negatives.

In the evaluations of identifying EMCPs and OCPs, I identify two issues that affect the effectiveness of the proposed 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 I 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 depen-

dencies 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. I plan to conduct experiments to evaluate the effectiveness of incorporating static analysis.

4.3 Contributions to Other Research

Besides identifying problems for DSE-based tools, the output of Covana can assist other test-generation and symbolic-analysis approaches, and the results of these research approaches can also improve the precisions of Covana.

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 [24] 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 [20, 21] can also be used to address the OCPs reported by Covana to increase coverage. Furthermore, loop summarization approaches [9, 17] may be employed to compute the summaries of the loops identified by BPs instead of all input-dependent loops, reducing the analysis space. The loop summarises can then be fed as inputs to the symbolic execution engine, which in turn improve the precision of our identification of BPs.

4.4 Future Works

I started working on the implementation of the techniques to identify BPs and already implemented the data dependence computation of partially-covered branch statements on simplified constraints. I plan to conduct evaluations of this new technique on collected benchmarks and complex programs in practice.

Visualization of Problems. To better assist developers in locating identified problems and providing guidances, I started a collaboration work with another fellow Ph.D student on developing a novel visualization approach. This ongoing work visualizes the structural coverage achieved by tools and the problem-analysis results produced by Covana, facilitating developers to understand how the reported problems affect the achieved coverage. We already conducted a user study involving 7 participants to evaluate the effectiveness of our visualization approach, and obtained promising results on the usefulness of our visualization. We also found insightful comments to improve the visualization approach through the user feedbacks.

Effective Error Messages. Good explanation of the identified problems are critical for developers to understand the identified problems. In fact, in our user study on the visualization approach, one user pointed out that our explanations of identified problems required improvement. I plan to further investigate how the work of Marceau et al. [11] can be employed to improve problem explanations to developers.

Other Directions for Cooperative Developer Testing. To finally realize cooperative developer testing, a deeper investigation on the the ways of the cooperation between developers and tools is required Humans are good at abstractive and intuitive tasks, such as specifying test oracles and local reasoning after a loop, while tools are good at mechanical and tedious tasks, such as test execution and test-input generation. I plan to leverage this synergy between humans and tools to further steer my research. As an example, the results of Thummalapenta et al.' work [20] show that the manual written tests achieved lower def-use coverage than the tests generated by tools, since def-use coverage may not be intuitive to measure when developers wrote the tests. Thus, it may be more beneficial to spend more efforts in developing the research of using tools to generate tests for achieving high def-use coverage.

5. REFERENCES

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

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

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

[4] C. Barrett and C. Tinelli. Cvc3. In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 298–302, Berlin, Heidelberg, 2007. Springer-Verlag.

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

[6] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proc. TACAS*, pages 337–340, 2008.

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

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

[9] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 23–33, New York, NY, USA, 2011. ACM.

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

[11] G. Marceau, K. Fisler, and S. Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 499–504, New York, NY, USA, 2011. ACM.

[12] Math.NET, 2008. `http://www.mathdotnet.com/`.

[13] C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .net with feedback-directed random testing. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 87–96, New York, NY, USA, 2008. ACM.

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

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

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

[17] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 225–236, New York, NY, USA, 2009. ACM.

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

[19] SvnBridge: Use TortoiseSVN with Team Foundation Server, 2009. `http://www.codeplex.com/SvnBridge`.

[20] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 189–206, New York, NY, USA, 2011. ACM.

[21] 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.

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

[23] N. Tillmann and W. Schulte. Parameterized Unit Tests. In *Proc. ESEC/FSE*, pages 253–262, 2005.

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

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

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

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