Precise Identification of Problems for Structural Test Generation

Xusheng Xiao[1] Tao Xie[1] Nikolai Tillmann[2] Jonathan de Halleux[2]
[1]Dept. of Computer Science, North Carolina State University, Raleigh, NC
[2]Microsoft Research, One Microsoft Way, Redmond, WA
[1]{xxiao2,txie}@ncsu.edu, [2]{nikolait,jhalleux}@microsoft.com

ABSTRACT

An important goal of software testing is to achieve at least high structural coverage. To reduce the manual efforts of producing such high-covering test inputs, testers or developers can employ tools built based on automated structural test-generation approaches. Although these tools can easily achieve high structural coverage for simple programs, when they are applied on complex programs in practice, these tools face various problems, such as (1) the external-method-call problem (EMCP), where tools cannot deal with method calls to external libraries; (2) the object-creation problem (OCP), where tools fails to generate method-call sequences to produce desirable object states. Since these tools currently could not be powerful enough to deal with these problems in testing complex programs in practice, 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 tools, in this paper, we propose a novel approach, called Covana, which 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. We provide two techniques to instantiate Covana to identify EMCPs and OCPs. Finally, we conduct evaluations on two open source projects to show the effectiveness of Covana in identifying EMCPs and OCPs.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA
Copyright 2011 ACM 978-1-4503-0445-0/11/05 ...$10.00

1. INTRODUCTION

Software testing is by far the most widely used technique for improving software reliability. An important goal of software testing is to achieve full or at least high structural coverage, such as statement or block coverage and branch coverage of the program under test. However, manually producing test inputs for achieving high structural coverage is labor-intensive. To address the issue, testers or developers can employ tools built based on state-of-the-art automated structural test-generation approaches to automatically generate test inputs, such as Dynamic Symbolic Execution (DSE) [3, 7] (also called concolic testing [7]).

Although these automated test-generation tools can easily achieve high structural coverage for simple programs, these tools face challenges in generating test inputs to achieve high structural coverage when they are applied on complex programs in practice. To better understand how automated test-generation tools perform for complex programs, we carried out a preliminary study of applying Pex [10], a DSE tool, on four popular open source projects, which have high download counts (study details are described in Section 2). The results show that the total block coverage achieved is 49.87%, with the lowest coverage being 15.54%. Among the problems that we empirically obervered, many statements or branches are not covered due to two major types of problems: (1) the external-method-call problem (EMCP), where method calls to external libraries[1] throw exceptions to abort test executions, or their return values are used to decide subsequent branches, causing the branches not to be covered; (2) the object-creation problem (OCP), where tools fail to generate sequences of method calls to construct desired object states for non-primitive method arguments or receiver objects to cover certain branches.

Since these automated tools could not be powerful enough to deal with various complicated situations in real-world code bases automatically without human intervention or guidance, we propose a new methodology of cooperative developer testing, where tools and developers cooperate to effectively carry out software testing as follows. Automated test-generation tools are first applied to generate test inputs and achieve coverage without human guidance. After the tools reach the pre-defined limits of resource consumption, the tools stop and report the achieved coverage and the problems that prevent them from achieving higher structural coverage back to developers, such as which external-method

[1]External libraries include native system libraries, such as file system and network socket libraries, and third-party pre-compiled libraries, where source code is not available.

call causes branches not to be covered or the state of which object is required to cover certain branches. By looking into the reported problems, the developers provide corresponding guidance to help the tools address the problems. For example, to deal with OCPs, developers can specify factory classes [10] that encode desired method sequences for non-primitive object types. To deal with EMCPs, developers can instruct the tools to instrument the external-method calls or provide mock objects [12] to simulate irrelevant environment dependencies. With the provided guidance, the tools are reapplied to generate test inputs for achieving higher structural coverage.

To achieve this cooperative developer testing, the tools need to report the encountered problems and narrow down the investigation scope, thus reducing the required efforts from the developers. Given the generated test inputs from tools and the achieved coverage, it is not difficult to identify the problem candidates for the developers to analyze. For example, locating all the external-method calls in the program under test can be easily achieved by static or dynamic program analysis, and reporting the object types of the program inputs and all their fields to the developers is fairly easy as well. However, the number of such problem candidates could be high, and quite some of these problem candidates (referred to as irrelevant problem candidates) are not causes for the tools not to achieve higher structural coverage. For example, the external-method call `Console.WriteLine` only prints the argument string value. Therefore, instrumenting or mocking this method call cannot result in any increase in coverage. Similarly, some branches may require only the specific object state of a field of the program inputs, and thus there is no need to spend efforts in providing sequences of method calls for all the fields of the program inputs.

Since the number of problem candidates could be large, the tools need to prune irrelevant problem candidates for reducing the efforts of developers in terms of investigation scope. Simple pruning techniques, such as pruning external-method calls by method names or belonging libraries, can result in high false positives and false negatives. In our preliminary study, we observed that if branch statements are data dependent on external-method calls for their return values (i.e., return values are used to decide which branches of the statements to take), the branches of these branch statements are very likely not covered by the generated test inputs, since automated test-generation tools normally cannot instrument or analyze the external-method calls. Hence, we can use compute data dependencies of branch statements containing not-covered branches (referred to as partially-covered branch statements) on external-method calls for their return values, and use the computed data dependencies to effectively identify such external-method calls and prune irrelevant ones. Similarly, we can compute data dependencies of partially-covered branch statements on program inputs and their fields, and use the computed data dependencies to help identify which fields of the program inputs require desired object states to cover certain not-covered branches.

To address the need of precisely identifying problems for developers to provide guidance, in this paper, we propose a novel approach called Covana that precisely identifies the problems that prevent the tools from achieving high structural coverage and prunes the irrelevant problem candidates using the data dependencies of partially-covered branch statements on problem candidates. 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 (including return values of external-method calls or program inputs as well as their fields) and perform forward symbolic execution [10] using test inputs generated by the tools as program inputs, (3) compute data dependencies of partially-covered branch statements on program candidates, and prune the candidates that none of partially-covered branch statements have data dependencies on. Since EMCPs and OCPs are the two major types of the problems observed in our preliminary study, we provide two specific techniques to instantiate Covana for identifying these two types of problems.

To show the effectiveness of Covana, in this paper, we use Dynamic Symbolic Execution (DSE) [3, 7, 10] as an illustrative example of automated structural-test-generation approaches. The primary reason why we choose DSE is that DSE is the most recent state-of-the-art in test generation. We concretize Covana as an extensible framework that collects information from DSE to identify different types of problem candidates and perform forward symbolic execution to compute data dependencies of partially-covered branch statements on problem candidates.

This paper makes the following major contributions:

- The first attempt to precisely identify problems faced by tools built for structural-test-generation approaches, achieving the first step of cooperative developer testing between tools and developers.

- A novel approach, called Covana, that identifies different types of problem candidates and prunes irrelevant candidates by computing data dependencies using forward symbolic execution.

- A concretization of Covana as a framework to identify problems that prevent DSE from achieving high structural coverage. Two analysis techniques are provided to instantiate Covana for precisely identifying EMCPs and OCPs.

- Two evaluations of Covana on two open source projects, including xUnit [14] and QuickGraph [6]. The results show that Covana effectively identifies 43 EMCPs out of 1610 EMCP candidates with only 1 false positive and 2 false negative, and 155 OCPs out of 451 OCP candidates with 20 false positives and 30 false negatives.

2. PRELIMINARY STUDY

In this section, we discuss the different types of problems that we empirically observed by applying a state-of-the-art DSE tool, Pex [10], on four open source projects for achieving structural coverage. The analysis of these problems helps motivate our approach. We choose Pex as the DSE tool in our empirical study and later we implement our approach upon it for two reasons: (1) Pex can explore all public methods of any real-world .NET code bases and generate test inputs automatically; (2) Pex has been applied internally in Microsoft to test core components of the .NET runtime infrastructure and found serious defects [10].

We apply Pex on the core libraries of the four open source projects until all the methods have been explored by Pex or

Project	LOC	Cov %	OCP	EMCP	Boundary	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

Pex runs out of memory and cannot continue to generate test inputs. These four open sources projects are SvnBridge [8], xUnit [14], Math.NET [4], and QuickGraph [6], which are quite popular and have high download counts. After Pex generates test inputs and produces coverage files, we select 10 source files that achieve low coverage in each project, and manually investigate the problems that contribute to the not-covered statements and branches. The details of the subjects and results can be found in our project web[2].

Table 1 shows the distribution of the problems that prevent DSE from achieving high structural coverage. 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 top major type of problems is object-creation problems (64.79%), shown in Column "OCP", since desired object states cannot be generated. In unit testing of object-oriented code, achieving high structural coverage requires desired object states for the receiver or non-primitive arguments of the method under test (MUT). These desired object states help cover various branches. However, automated approaches are often ineffective in generating method-call sequences that produce desired object states to achieve high structural coverage [9], facing object-creation problems that prevent DSE from achieving high structural coverage.

The second major type of problems is external-method-call problems (26.76%), shown in Column "EMCP", since external-method calls cause to lose track of computed symbolic values passed as their arguments or throw exceptions to hinder the exploration. In our 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 we simply report every encountered external-method call as an EMCP, we can get many irrelevant problems that are not cause for any not-covered statment or branch.

The third main type of problems is boundary problems (5.63%), shown in Column "Boundary", 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. The reason why we did not have so many not-covered branches due to this type of problems is that the used constraint solver generates approximate integers for constraints that contain floating-point arithmetics and these approximate integers can cover certain branches.

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

```
static string GetDefaultConfigFile(string assembly-
File)  {
00:  string configFilename = assemblyFile + ".config";
01:  if (File.Exists(configFilename))
02:    return configFilename;
03:  return null;
04:  }
...
public ExecutorWrapper(string assemblyFilename, ...) {
05:  ...
06:  assemblyFilename = Path.GetFullPath(assemblyFilename);
07:  ...
}
public AssertActualExpectedException
              (object expected, object actual, ...) {
08:  ...
09:  this.actual += String.Format("(0)",
                              actual.GetType().FullName);
10:  this.expected += String.Format("(0)",
                              expected.GetType().FullName);
11:  ...
}
```

Figure 1: Three simplified methods from xUnit [14].

3. EXAMPLE

We next explain how Covana, instantiated with two specific techniques, identifies EMCPs and OCPs with two illustrative examples.

3.1 External-Method-Call Problem (EMCP)

During the execution of the automatically generated test inputs, external-method calls may prevent the generated test inputs from achieving high structural coverage if the return values of external-method calls are used to decide subsequent branches to take or throw exceptions to terminate test executions. As a real example, the return value of `File.Exists` in Figure 1 is used in deciding which branch at Line 1 to take. If the generated test inputs do not contain a file name that exists in the test environment, being mostly the case, the statement at Line 2 cannot be covered by the test inputs. The method `Path.GetFullPath` at Line 6 is another example external-method call that prevents test inputs from achieving higher structural coverage. `Path.GetFullPath` throws exceptions when an invalid of an assembly file is given as the argument. Therefore, if none of the generated inputs includes a valid name, the lines after Line 6 remain not-covered. However, not all the external-method calls can cause problems for achieving high structural coverage. For instance, `String.Format` at Line 9 and 10 in Figure 1 only formats the string value of the input and does not affect the coverage achieved by the generated test inputs.

Covana first identifies as problem candidates the external-method calls whose arguments have data dependencies on program inputs. In Figure 1, Covana identifies `File.Exists` at Line 1, `Path.GetFullPath` at Line 6, and `String.Format` at Lines 9 and 10 as candidates, since they all have data dependencies on the program inputs. By assigning symbolic values to return values of the candidates and applying forward symbolic execution [3, 7], Covana collects symbolic expres-

```
public class FixedSizeStack {
00:  private Stack stack;
01:  public FixedSizeStack(Stack stack) {
02:    this.stack = stack;
03:  }
04:  public void Push(object item) {
05:    if(stack.Count() == 10) {
06:      throw new Exception("full");
07:    }
08:    stack.Push(item);
09:  }
10:  ...
}
11:  public void TestPush(FixedSizeStack stack,
                           object item){
12:    stack.Push(item);
13:  }
```

Figure 2: FixedSizeStack implemented using Stack

sions in the predicates of branch statements. From the symbolic expressions collected from branch statements, Covana extracts elements of problem candidates and considers the branch statements that have data dependencies on problem candidates. If one of the branches at Line 1 is not covered (i.e., Line 1 is a partially-covered branch statement), Covana identifies `FileExists` as an EMCP. On the other hand, there are no partially-covered branch statements that are data dependent on the return values of `String.Format` at Lines 9 and 10, causing `String.Format` at Lines 9 and 10 to be pruned. For `Path.GetFullPath`, if all of its executions throw exceptions, the remaining part of the program, starting at Line 7, remains not-covered. Covana detects the exceptions that cause to abort the test executions and identifies `Path.GetFullPath` as an EMCP that causes the area starting at Line 7 not to be covered.

3.2 Object-Creation Problem (OCP)

Figure 2 shows a class `FixedSizeStack` that has a field `stack` of type `Stack`. Invoking the method `Push` to push objects is required for increasing the size. `Stack` has a field `items` that stores the pushed objects, and the method `stack.Count()` returns the number of objects stored in the `Stack.items`[3]. `FixedSizeStack` has an upper bound of the number of objects that can be pushed into the stack. To bound the size, the method `FixedSizeStack.Push` throws an exception when the size of the stack has reached the bound (10 in the example). The method `TestPush` receives a `FixedSizeStack` object and an object to be pushed as its arguments and invokes the method `FixedSizeStack.Push` to push the object to the stack for testing. To cover the true branch at Line 5 of the method `FixedSizeStack.Push`, the generated test inputs need to include method-call sequences to create a full `FixedSizeStack` whose size is 10.

Since the field `FixedSizeStack.stack` can be assigned directly by invoking the constructor of `FixedSizeStack` and passing an object of `Stack` as an argument (i.e., the field `FixedSizeStack.stack` is *assignable* for the declaring class `FixedSizeStack`), the difficulty of generating an object state of `FixedSizeStack` whose size is 10 lies in generating an object of `Stack` whose size is 10. Let us assume that automated test-generation tools cannot produce the required object state of `Stack`.

Covana first assigns symbolic values to program inputs and their fields, i.e., `FixedSizeStack`, `FixedSizeStack.stack`,

[3]Assume that `Stack.items` is implemented using the object type `List<object>`

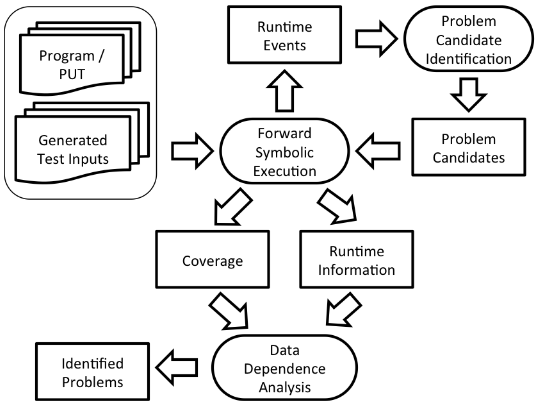


Figure 3: Overview of Covana

and `Stack.items`, and performs forward symbolic execution to compute data dependencies. By computing data dependencies of partially-covered branch statements, Covana figures out that the branch statement at Line 5 (with the true branch not-covered) has data dependencies on `Stack.items`. However, reporting the object type of `Stack.items`, being `List<object>`, results in a false warning. Since by providing method-call sequences for `List<object>`, the tools cannot assign it to the field `Stack.items` since `Stack.items` is assignable for `Stack`.

Based on this observation, Covana constructs a field declaration hierarchy from the field that the branch statement has data dependencies on up to the program input and identifies the declaring class whose field is not assignable as the cause of the OCP. In this example, the constructed hierarchy is `FixedSizeStack`, `FixedSizeStack.stack`, and `Stack.items`. Covana analyzes the field declaration hierarchy starting from the program input. By analyzing `FixedSizeStack` and `FixedSizeStack.stack`, Covana knows that `FixedSizeStack.stack` is assignable for `FixedSizeStack`. Then Covana continues to check `FixedSizeStack.stack` and `Stack.items`. Since the field `Stack.items` can be changed only by invoking the method `Stack.Push` (i.e., not assignable for `Stack`), Covana identifies the object type `Stack` of `FixedSizeStack.stack` as an OCP that causes the true branch at Line 5 not to be covered.

4. APPROACH

In this section, we describe how Covana identifies problem candidates of structural test generation and prunes irrelevant problem candidates by computing data dependencies. Covana consists of three main steps: Problem-Candidate Identification, Forward Symbolic Execution, and Data Dependence Analysis. In the following part of the section, we introduces the overview of Covana and describe these three main steps in detail.

4.1 Overview of Covana

In this paper, we concretize Covana as an extensible framework for identifying problems that prevent DSE from achieving high structural coverage. DSE [3, 7] executes the program symbolically, starting with arbitrary inputs. Along the execution path, DSE collects symbolic constraints on program inputs in branch nodes (being runtime instances of branch statements) to form an expression, called the path condition. To obtain a new path that takes a different branch, one of the branch nodes in the path condition is negated to create a new path condition that shares the prefix up till the node being negated with the original path. Then a constraint solver is used to compute test inputs that satisfy the new path condition. These generated test inputs again are executed on the program to explore different paths of the program. Ideally, all feasible paths can be exercised eventually through such iterations of path variations. However, as we discussed in the introduction, various problems cause DSE not to achieve high structural coverage.

Figure 3 shows a high-level overview of Covana. Covana accepts as input a program under test or Parameterized Unit Test (PUT) [11], and generated test inputs from automated test-generation tools (such as a DSE-based tool). Covana then leverages the DSE engine to perform forward symbolic execution on the program or PUT using the test inputs as program inputs (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, 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, and prunes irrelevant problem candidates that none of partially-covered branch statements have data dependencies on. In our current prototype, we instantiate this general approach with two techniques to identify the top two main types of problems: EMCPs and OCPs. We next discuss each step of Covana in detail.

4.2 Problem-Candidate Identification

Covana collects necessary information from the DSE engine for identifying different types of problem candidates. The DSE engine executes the program under test or PUT with the generated test inputs symbolically. During execution, Covana monitors different events triggered by the DSE engine and exposes these events as interfaces for specifying different types of problem candidates. There are many kinds of events that can be monitored, such as events of method entry and method exit. We next discuss how these events can be used to identify the problem candidates of EMCPs and OCPs.

4.2.1 Identifying EMCP Candidates

A method exit event is triggered by DSE engine when the execution of a method call is finished. This event comes with detailed method information, including method arguments, method instrumentation information, and so on.

If the method is not instrumented by DSE, the method call is considered as an external-method call, method calls to either system libraries or third-party pre-compiled libraries. If Covana considers all external-method calls as problem candidates of EMCP, then the number of problem candidates can be very large for complex programs. Hence, Covana considers as candidates only the external-method calls whose arguments have data dependencies on program inputs. In this way, the external-method calls that have constant arguments are not considered as problem candidates and are pruned without computing data dependencies. Normally, such external-method calls are method calls that print constant strings or put a thread to sleep for some time, which do not cause DSE not to achieve higher structural coverage and can be safely pruned. Since DSE typically assigns symbolic values to program inputs, to know whether method arguments have data dependencies on program inputs can be achieved easily by checking whether the method arguments contain symbolic expressions of program inputs.

In our preliminary study described in Section 2, we observed that many branches are not covered since the conditions of these branches use the return values of external-method calls (i.e., data dependent on these external-method calls). The reason is that DSE tools and other automated test-generation tools are unlikely to generate different test inputs to cause external-method calls to return desired values since these tools have not instrumented or analyzed external-method calls. Therefore, for the external-method calls whose arguments have data dependencies on program inputs, Covana considers them as EMCP candidates. To illustrate the analysis, we use the example shown in Figure 1. During the test execution of the method `GetDefaultConfigFile`, Covana identifies the external-method call `File.Exists` as an EMCP candidate, since its argument `configFilename` has data dependency with `assemblyFile`, which is the program input. The return value of `File.Exists`, which is used in the branch statement at Line 1, is assigned with a symbolic value for computing data dependencies.

4.2.2 Identifying OCP Candidates

Whenever test inputs are used as the arguments to execute the program under test, the method entry event of the method under test is triggered. In this exposed event, the details of the generated program inputs are collected. Since OCP requires objects of a non-primitive type as program inputs, Covana ignores program inputs whose type is primitive type, such as `int`, `double`, and `boolean`. Covana considers the program inputs of non-primitive types themselves and their fields of non-primitive types as OCP candidates.

4.3 Forward Symbolic Execution

Covana performs forward symbolic execution using the test inputs generated by automated test-generation tools as program inputs, and collects runtime information for computing data dependencies. Covana assigns symbolic values to elements of the identified problem candidates (such as return values of external-method calls) and leverages the DSE engine to perform forward symbolic execution for collecting constraints on elements of problem candidates in branch statements. We next discuss how Covana uses this runtime information to compute data dependencies of partially-covered branch statements on problem candidates.

4.3.1 Collecting Symbolic Expressions in Branches

Since elements of problem candidates are assigned with symbolic values, if a branch statement has data dependency on problem candidates, we can find symbolic expressions (on elements of the problem candidates) in the predicates of the branch statement. Such information is later used to compute data dependencies on problem candidates.

4.3.2 Collecting Uncaught Exception

After assigning symbolic values to elements of problem candidates, Covana monitors the program execution. Whenever an uncaught exception is thrown, Covana collects the exception including its stack trace of the exception. As we observed in the preliminary study, if an external-method call throws an exception for the executions of all the generated test inputs, the remaining parts of the program after the call site of the external-method call cannot be covered. Thus, Covana uses the stack trace of an exception thrown at runtime for the analysis of EMCP described in Section 4.4.1.

4.4 Data Dependence Analysis

Covana consumes the collected runtime information from the forward symbolic execution to compute data dependencies. For each collected symbolic expression *sym* found in the predicates of a branch statement b, Covana extracts elements of the problem candidates *elem* from *sym*. From *elem*, Covana extracts the corresponding problem candidates P and considers b has data dependency on P. Using the collected structural coverage, Covana further computes data dependencies of partially-covered branch statement on problem candidates. With these data dependencies, different analyses further prune irrelevant problem candidates.

4.4.1 EMCP Analysis

Covana first identifies EMCP using the data dependencies of partially-covered branch statements on EMCP candidates. If these exist some partially-covered branch statements that have data dependencies on EMCP candidates for their return values, Covana directly reports such external-method calls as EMCPs. To identify external-method calls that throw exceptions to abort test executions, Covana further analyzes the method calls from the collected stack traces of exceptions thrown during runtime. If these method calls contain any external-method call and the remaining parts of the program after the call site of the external-method call are not covered, Covana identifies the extracted external-method call as an EMCP that causes the remaining parts of the program not to be covered.

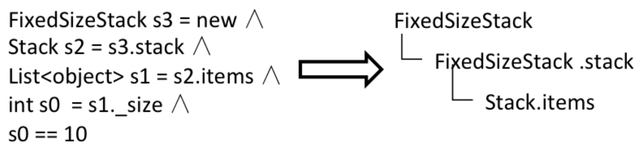
Algorithm 1 Object Creation Problem (OCP) Analysis

```
Require: $Fields$ for field declaration hierarchy, $B$ for not-covered branches
Ensure: $OCP$
 1: if $Length(Fields) == 1$ then
 2:     $OCP = CreateOCP(TypeOf(Fields[0]), B)$
 3:     return $OCP$
 4: else
 5:     Set $current = NULL$
 6:     for $i = 1$ to $Length(Fields) - 1$ do
 7:         $current = Fields[i]$
 8:         $dc = TypeOf(Fields[i-1])$
 9:         $assg = IsAssignable(current, dc)$
10:         if $!assg$ then
11:             $OCP = CreateOCP(dc, B)$
12:             return $OCP$
13:         end if
14:     end for
15:     $OCP = CreateOCP(TypeOf(current), B)$
16:     return $OCP$
17: end if
```

4.4.2 OCP Analysis

Covana identifies OCPs using the data dependencies of partially-covered branch statements on program inputs and their fields. If a partially-covered branch statement is data dependent on only program inputs, Covana directly reports the program inputs as OCPs. However, if a partially-covered branch statement is data dependent on fields of program inputs, Covana constructs a field declaration hierarchy up to a program input and performs further analysis to identify which field causes tools not to achieve high structural coverage.

To construct a field declaration hierarchy of the field f that a partially-covered branch statement is data dependent on, we can use reflection to obtain the class structure of a program input p and search the fields for finding f (i.e., f is one of the fields of p). If we fail to find f, we continue to search the class structures of the fields of p, similar to graph searching. The search continues until we find f. The fields along the path from p to f are used to construct the field declaration hierarchy. Another way, which Covana adopts, is to use path conditions that lead to not-covered branches and extract fields directly from path conditions, since the symbolic expressions in the path conditions already contain program inputs and their fields. To illustrate the extraction, we use the example shown in Figure 2. The figure below shows the path condition that leads to the true branch at Line 5 and the field declaration hierarchy constructed from the path condition.



```
FixedSizeStack s3 = new ∧
Stack s2 = s3.stack ∧
List<object> s1 = s2.items ∧
int s0  = s1._size ∧
s0 == 10
```

Algorithm 1 shows our algorithm that identifies OCPs by analyzing the field declaration hierarchy (Fields) and the not-covered branches (B). If the extracted field declaration hierarchy contains only one field (satisfying Line 1 of the algorithm), which should be the program input itself, our algorithm reports the program input as an OCP directly.

To identify which field in the field declaration hierarchy causes the OCP, our algorithm analyzes the field declaration hierarchy level by level, starting from Level 2 (i.e., the field of the program input). If a field is assignable for its declaring class (not satisfying Line 10), DSE or other automated test-generation tools can easily create an object of the field's class type and assign the object to the field by invoking the corresponding constructor or public setter method. In this case, it is the object type of the field or a field in the next level(s), not its declaring class, that causes an OCP. To further decide whether it is the current field or the field in the next level that causes the OCP, our algorithm then continues to check the field in the next level (back to Line 7).

If a field is not assignable for its declaring class (satisfying Line 10), the object state of the field can be changed only by invoking other public state-modifying methods of its declaring class. Hence, Covana reports the type of its declaring class as an OCP. In our example shown in Figure 2, the field `Stack.items` cannot be assigned with an object of `List<object>` by invoking any constructor or public setter method of `Stack`. To change the object state of `Stack.items`, DSE needs specific sequences of method calls for `Stack` instead of `List<object>`. As a result, Covana identifies the object type `Stack` as an OCP (Line 11).

5. IMPLEMENTATION

The prototype implementation of Covana includes three parts: (1) the extension to Pex [10], an automatic white-box test generation tool built for DSE. This extension identifies problem candidates, assigns symbolic values to elements of problem candidates, and collects runtime information; (2) a data-dependence analyzer that analyzes the information produced by these two components and identifies EMCPs and OCPs; (3) a Graphic User Interface (GUI) component that shows the identified problems with detailed analysis information.

6. EVALUATIONS

In this section, we discuss the two evaluations conducted to show the effectiveness of Covana. In our evaluations, we use two popular .NET applications: xUnit [14] and QuickGraph [6], and answer the following research questions:

- **RQ1**: How effective is Covana in identifying the two main types of problems, EMCPs and OCPs?
- **RQ2**: How effective is Covana in pruning irrelevant problem candidates of EMCPs and OCPs?

We next provide details of the metrics that we collect in our evaluations. To measure the effectiveness of our approach in identifying EMCPs and OCPs (addressing RQ1), we measure the number of problems that Covana finds for the not-covered branches or statements of the subject applications. To measure the effectiveness of our approach in pruning irrelevant problem candidates (addressing RQ2), we compare the number of identified problem candidates with the number of identified problem by our approach in applications under test and measure the number of problem candidates pruned by our approach. To address both RQ1 and RQ2, we measure the false positives, i.e., the number of irrelevant problem candidates that are not pruned by Covana, and the false negatives, i.e., the number of real problems that are identified as irrelevant problem candidates and pruned.

We next provide details on the subject applications and evaluation setup, and the results of the two evaluations.

6.1 Subjects and Evaluation Setup

We used two popular .NET applications for evaluating our Covana approach: xUnit [14] and QuickGraph [6]. xUnit is a unit testing framework for .NET program development. xUnit includes 223 classes and interfaces with 11.4 KLOC. QuickGraph is a C# graph library that provides various directed and undirected data structures of graphs. QuickGraph also provides graph algorithms such as depth-first search, topological sort, and shortest path [2]. QuickGraph includes 165 classes and interfaces with 8.3 KLOC.

In our evaluations, we use Pex with the implemented extensions as our DSE test-generation tool. The Pex version used for our evaluation is 0.24.50222.1. We first apply Pex to explore the applications under test and generate test inputs. After test generation and execution, which is automated by Pex, the coverage and the collected runtime information are fed into our stand-alone analysis tool for identifying EMCPs and OCPs.

We next discuss the results of our evaluations in terms of the effectiveness of Covana in identifying EMCPs and OCPs, and in reducing the irrelevant problem candidates.

6.2 RQ1: Problem Identification

In this section, we address the research question RQ1 of how effectively Covana identifies EMCPs and OCPs. To address this question, we measure the number of identified problems, the number of false positives, and the number of false negatives generated by Covana. To measure values for these metrics, we executed the stand-alone analysis tool implemented for our approach with the output information from Pex as inputs, and manually classified the problems reported by our tool as real problems, false positives, and false negatives. To verify EMCP candidates, we either instrument or provide mock objects for the external-method calls identified as EMCP candidates, and reapplied Pex to check whether the not-covered branches can be covered. If so, we classify the EMCP candidates as real problems, or irrelevant problem candidates otherwise. Similarly, to verify OCP candidates, we provide sequences of method calls for the object types of the OCP candidates, and reapplied Pex to check whether the not-covered branches can be covered. If so, we classify the OCP candidates as real problems, or irrelevant problem candidates otherwise.

Table 2 shows the results for all the assemblies in both subject applications. Column "# File" lists the number of source files in each application assembly. Columns "Object-Creation Problem (OCP)" and "External-Method-Call Problem (EMCP)" show the statistics of EMCPs and OCPs identified by Covana. Subcolumn "# Real" gives the number of real problems identified by us manually. Subcolumn "Identified" gives the number of problems identified by Covana, and subcolumn "# FP" and "# FN" give the number of false positives and false negatives, respectively. The results show that our approach identifies 43 EMCPs with only 1 false positive and 2 false negatives. In addition, our approach identifies 155 OCPs with 20 false positives and 30 as false negatives. The reason why we have 30 false negatives is that in our prototype analysis tool, we did not implement the logics required to handle `Dictionay` objects (C# version of `HashMap`) and static fields of classes. In our future work, we plan to address these issues by identifying the fields of `Dictionay` objects and static fields of classes as candidates and computing data dependencies of partially-covered branch statements on them.

We next provide examples to describe scenarios where our approach effectively identifies EMCPs and OCPs. We also describe scenarios where our approach produces false positives and false negatives.

Figure 4 shows the class `TestClassCommand` of the `Xunit.Sdk` namespace. When we applied Pex to generate test inputs for the method `TestClassCommand.ClassStart`, Pex generated only one test input and achieved low block coverage of 2/27 (7.14%). In `TestClassCommand.ClassStart`, the loop at Line 8 requires the field `TestClassCommand.typeUnderTest` to be not null. Since Pex cannot find in the application any public class that implements the interface `ITypeInfo` to create such an object for `TestClassCommand.typeUnderTest`, Pex cannot generate more useful test inputs. Thus, we need to report an OCP of the interface type `ITypeInfo`. By analyzing the data dependencies of the entry branch of the loop at Line 8, our approach extracts the argument object `TestClassCommand` and its field `TestClassCommand.typeUnderTest`. By analyzing `TestClassCommand` and `TestClassCommand.typeUnderTest`, our approach figures out that `TestClassCommand.typeUnderTest` can be assigned by using the public constructor of the class

Application Assembly	# File	Object-Creation Problem (OCP)				External-Method-Call Problem (EMCP)			
		# Identified	# Real	# FP	# FN	# Identified	# Real	# FP	# FN
xUnit	71	68	67	13	12	24	24	0	0
xUnit.Extensions	17	7	5	3	1	2	2	0	0
xUnit.Console	7	2	2	0	0	2	2	0	0
xUnit.Gui	12	3	3	0	0	1	3	0	2
xUnit.Runner.Msbuild	6	15	14	1	0	0	0	0	0
xUnit.Runner.Tdnet	3	5	5	0	0	1	1	0	0
xUnit.Runner.Utility	28	7	12	0	5	9	9	0	0
Quickgraph	3	0	0	0	0	0	0	0	0
Quickgraph.Algorithms	12	7	11	0	4	0	0	0	0
Quickgraph.Algorithms.Graphviz	14	20	20	2	2	4	3	1	0
Quickgraph.Collections	19	6	11	1	6	0	0	0	0
Quickgraph.Concepts	35	5	5	0	0	0	0	0	0
Quickgraph.Exceptions	3	0	0	0	0	0	0	0	0
Quickgraph.Predicates	9	8	8	0	0	0	0	0	0
Quickgraph.Representations	3	2	2	0	0	0	0	0	0
Total	242	155	163	20	30	43	44	1	2

Table 2: Evaluation results showing the effectiveness of Covana in identifying EMCP and OCP

```
public class TestClassCommand : ITestClassCommand {
00:  readonly Dictionary<MethodInfo, object> fixtures
       = new Dictionary<MethodInfo, object>();
01:  Random randomizer = new Random();
02:  ITypeInfo typeUnderTest;
03:  ...
04:  public TestClassCommand(ITypeInfo typeUnderTest) {
05:    this.typeUnderTest = typeUnderTest; }
06:  public Exception ClassStart() {
07:    try {
08:      foreach (Type @interface in
              typeUnderTest.Type.GetInterfaces()) {
09:        ...
10:      }
11:    }
12:  ...
13:  }
14:  public Exception ClassFinish() {
15:    foreach (object fixtureData in fixtures.Values) {
16:      ...
17:    }
18:  }
}
```

Figure 4: `TestClassCommand` class of xUnit

`TestClassCommand`, and correctly reports an OCP of `ITypeInfo`.

Similarly, Pex achieves low coverage block coverage of 6/16 (37.50%) when generating test inputs for the method `TestClassCommand.ClassFinish`. The reason is that the loop at Line 15 requires the field `TestClassCommand.fixtures` to hold at least one item. Since there is no constructor or public setter method to assign an external object to `TestClassCommand.fixtures`, other public methods of `TestClassCommand` need to be invoked to change the value of `TestClassCommand.fixtures`. Therefore, we need to report the program input `TestClassCommand` as an OCP. However, our approach cannot detect such situation since the object type of the field `fixtures` is `Dictionary` and we did not implement the logics to handle such type. Hence, our approach did not identify the object type of the `fixtures` as an OCP for the not-covered branch at Line 15.

Figure 5 shows two methods: (1) the method `ParseCommandLine` of class `Program` in the namespace `Xunit.ConsoleClient` and (2) the constructor of the class `Executor` in the namespace `Xunit.Sdk`. For `ParseCommandLine`, Pex achieved low block coverage of 44/154 (28.57%), because it cannot generate test inputs to cause the external-method call `File.Exists` to return true. Since the `out` variable `assemblyFile` is assigned with the value of `args[0]` (and thus has data dependencies on the program input `args[0]`), our approach assigned a symbolic value to the return value of `File.Exist` and found that the branch statement at Line 1 (the false branch not-covered) has data dependency on `File.Exist` for its return value. Thus, our approach correctly reported an EMCP of `File.Exists`. For the constructor of the class `Executor`, Pex achieved low block coverage of 2/5 (40%), because Pex generated a `null` object as the argument for the constructor, which caused the external-method call `Path.GetFullPath` to throw an exception. Our approach collected this exception thrown from `Path.GetFullPath` during runtime. By checking the coverage of the remaining parts of the program after the call site of `Path.GetFullPath`, our approach found that none of them was covered. As a result, our approach reported `Path.GetFullPath` as an EMCP. Although another external method `Console.WriteLine` at Line 2 receives `assemblyFile` as argument and is marked as an EMCP candidate by our approach, this external method did not have any return value, and thus no branch statements have data dependencies on `Console.WriteLine`. As a result, our approach correctly pruned `Console.WriteLine`.

```
static bool ParseCommandLine(string[] args,
                   out string assemblyFile, ...) {
00:  assemblyFile = args[0];
...
01:  if (!File.Exists(assemblyFile)) {
02:    Console.WriteLine("error: assem-
bly file not found: {0}", assemblyFile);
03:    return false;
04:  }
...
public Executor(string fileName) {
05:  this.assemblyFilename = Path.GetFullPath(fileName);
06:  ...
}
```

Figure 5: Two methods that have EMCPs in xUnit

6.3 RQ2: Irrelevant-Problem-Candidate Pruning

In this section, we address the research question RQ2 of how effectively our approach prunes irrelevant problem candidates. To address this question, we compare the number of identified problem candidates with the number of problems reported by our approach, and measure the number of

Application	Object-Creation Problem (OCP)					External-Method-Call Problem (EMCP)				
	#Cand	#Ident	#Pruned	#FP	#FN	#Cand	#Ident	#Pruned	#FP	#FN
xUnit	335	107	228 (68.06%)	17	18	1313	39	1274 (97.03%)	0	2
QuickGraph	116	48	68 (58.62%)	3	12	297	4	293 (98.65%)	1	0
Total	451	155	296 (65.63%)	20	30	1610	43	1567(97.33%)	1	2

Table 3: Evaluation results showing the effectiveness of Covana in reducing irrelevant problem candidates

```
// Lines 1, 2, 4 are external-method calls
public static List<RecentlyUsedAssembly> LoadAssem-
blyList() {
00:  ...
01:  using (var xunitKey = Registry.CurrentUser.
            CreateSubKey(XUNIT_KEY_NAME))
02:  using (var recentKey = xunitKey.
            CreateSubKey(RECENT_ASSEMBLIES_KEY_NAME)){
03:    for (int index = 0; ; ++index)
04:      using (var itemKey = recentKey.
                  OpenSubKey(index.ToString())) {
05:      if (itemKey == null) {
06:        break;
07:      }
08:      if (itemKey != null) {
09:        ...
10:      }
11:    }
12:  }
}
```

Figure 6: The method `LoadAssemblyList` in the class `RecentlyUsedAssemblyList` of xUnit

problem candidates pruned by our approach. In addition, we measure the false positives, i.e., the irrelevant problem candidates not pruned by Covana, and the false negatives, i.e., the real problems pruned by Covana. To measure values for these metrics, we executed the stand-alone analysis tool implemented for our approach with the output information from Pex as inputs, and manually classified the problem candidates reduced by our tool as real problems, false positives, and false negatives in the same way as in addressing RQ1.

Table 3 shows the results of both subject applications. Column "Application" lists the names of the subject applications. Columns "External-Method-Call Problem" and "Object-Creation Problem" show the statistics of EMCPs and OCPs, respectively. Here, the EMCP candidates are all the encountered external-method calls during the test execution, and the OCP candidates are all the non-primitive object types of program inputs and their fields that DSE assigns symbolic values to. In Table 3, subcolumn "#Cand" gives the number of problem candidates, subcolumn "#Ident" gives the number of problems identified by Covana, and subcolumns "#FP" and "#FN" give the number of false positives and false negatives, respectively. The results show that our approach prunes 97.33% (1567 in 1610) EMCP candidates with only 1 false positive and 2 false negatvies and prunes 65.63% (296 in 451) OCP candidates with 20 false positives and 30 false negatives. These results show that our approach effectively reduces the irrelevant problem candidates with low false positives and false negatives.

7. DISCUSSION AND FUTURE WORK

Covana identifies problems faced by tools built for structural test-generation approaches and prunes irrelevant problem candidates to reduce the problem space for investigation. Covana serves as the first step towards problem solving. In fact, identifying problems for developers to investigate is analogical to fault localization before fault fixing. Below, we discuss how Covana can be used to assist other automated test-generation approaches or manual test-generation approaches, and then discuss some issues including those encountered in our evaluations.

Assisting Other Structural Test-Generation Approaches. Given test inputs, no matter whether they are generated by other automated test-generation approaches, such as a random approach, or are generated manually, Covana can be used to identify problems of specific types, such as EMCPs and OCPs. The analysis result of Covana not only can reduce the efforts of developers in providing guidance to tools, but also can reduce the cost of tools built for other test-generation approaches. The first example is to automatically generate mock objects for only the external-method calls identified as EMCPs by Covana. Since Covana greatly reduces the number of irrelevant problem candidates of EMCP, it becomes possible to generate mock objects for the external-method calls identified as EMCPs. As another example, random approach can assign more probabilities on exploring the object types reported as OCPs by Covana, increasing the chances to achieve higher structural coverage in shorter time. Advanced method-sequence-generation approaches [9] can also be used to address OCPs for increasing coverage.

Static Field. In our evaluations, we observed that a few classes contained static fields that were initialized inside the classes. These static fields were later used by some branches and some of these branches were not covered by DSE. Since DSE did not automatically assign symbolic values to static fields, DSE was not able to collect symbolic constraints on these static fields. In future work, we plan to assign symbolic values to these static fields, so that our approach can collect the symbolic constraints on these static fields for our analysis.

Concrete Arguments for External-Method Calls. Our current Covana implementation identifies the return values of external methods as candidates if the method arguments have data dependencies on program inputs. However, in our evaluations, there were a few external-method calls received concrete values as arguments, and resulted in some not-covered branches. The external-method call `recentKey.OpenSubKey`, shown in Figure 6, received a concrete value returned by `index.ToString()`. Since its return value `itemKey` of `recentKey.OpenSubKey` is `null`, the false branch at Line 5 is not covered. In this case, our approach cannot detect the problem, since our approach does not mark as a candidate the return value of any external-method call that does not receive any symbolic values as an argument. By assigning symbolic values to all external-method calls, our approach can be easily extended to compute data dependencies on every external-method call, no matter whether its arguments have data dependencies on program inputs. However, computing data dependencies on every external-method call may incur many false positives and increase the performance overhead significantly, since the number of external-method calls encountered during the program executions is not trivial. In future work, we plan to conduct experiments to measure the effectiveness and performance overhead when every external-method call is considered as a candidate.

Other Potential Issues. Besides the issues encountered 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 that 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, and 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 detected data dependencies, which would compromise the effectiveness of our approach. We plan to conduct experiments to evaluate the effectiveness of incorporating argument side effect, control dependency, and static analysis.

8. RELATED WORK

Coverage Analysis. Pavlopoulou and Young [5] developed a residual coverage monitoring tool for Java, which provides richer feedback from actual use of deployed software. Since their approach aims to reduce the performance overhead for gathering structural coverage from deployed software, their approach does not provide a way to analyze the coverage, while our approach analyzes the residual structural coverage gathered from DSE to filter out irrelevant problem candidates.

Explaining Failures of Program Analysis. Dincklage and Diwan [13] propose an analysis language and build a system to produce reasons when program analyses fail to produce desirable results. The objective of their approach is to express arbitrary data flow analyses using their analysis language and compute reasons for the failures. 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 quite different problem and includes significantly different techniques needed for addressing unique challenges in identifying problems that prevent test-generation tools from achieving high structural coverage.

Symbolic Execution. Anand et al. [1] propose type-dependence analysis, which performs a context- and field-sensitive interprocedural static analysis to identify the parts of the program under test that may be unsuitable for symbolic execution, such as third-party libraries. Their approach identifies external-method calls that are problematic 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 data dependencies of arguments of external-method calls on program inputs, but also data dependencies of partially-covered branch statements on external-method calls for their return values.

9. CONCLUSION

In this paper, we propose cooperative developer testing, where developers provide guidance to help structural test-generation tools achieve high structural coverage. To reduce the efforts of developers in providing guidance, we propose a novel approach, called Covana, which precisely identifies and reports problems that cause structural test-generation tools not to achieve high structural coverage. Covana identifies these problems by computing data dependencies of partially-covered branch statements on problem candidates. We concretize Covana to identify problems faced by DSE and present two techniques to identify EMCPs and OCPs, the top two major types of problems. We also evaluate Covana on two open source projects and the results show that Covana effectively identifies EMCPs and OCPs.

Acknowledgments. This work is supported in part by NSF grants CNS-0716579, CCF-0725190, CCF-0845272, CCF-0915400, CNS-0958235, an NCSU CACC grant, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU SOSI.

10. REFERENCES

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

[2] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

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

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

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

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

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

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

[9] 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*, pages 193–202, 2009.

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

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

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

[13] D. von Dincklage and A. Diwan. Explaining failures of program analyses. In *Proc. PLDI*, pages 260–269, 2008.

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