Issue Analysis for Residual Structural Coverage in Dynamic Symbolic Execution

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

ABSTRACT

The process of achieving high structural coverage of the program under test can be automated using Dynamic Symbolic Execution (DSE), which generates test inputs to iteratively explore paths of the program under test. When applied on real-world applications, DSE faces various challenges in generating test inputs to achieve high structural coverage. Among issues related to these challenges, our preliminary study identified two main types of issues: (1) object-creation issues (OCI), where DSE fails to generate method-call sequences to produce desirable object states; (2) external-method-call issues (EMCI), where symbolic values are passed as arguments to third-party library methods that are not instrumented by DSE. Automatically solving these two main types of issues is challenging, since the exploration space of generating method-call sequences for desirable object states is usually too huge, and instrumenting all third-party libraries can cause explosion of the exploration space. However, when provided with informative information of issues, users can effectively assist DSE to achieve high structural coverage. In this paper, we propose a general approach, called Covana, to identify issues faced by DSE via analyzing runtime information, and filter out irrelevant issues using residual structural coverage. We provide two techniques to instantiate our general approach to identify OCIs and EMCIs. To show the effectiveness of Covana, we conduct evaluations on two open source projects. Our results show that Covana effectively identifies 155 OCIs, and 43 EMCIs. Moreover, Covana effectively reduces 296 irrelevant issues out of 451 OCIs and 1567 irrelevant issues out of 1610 EMCIs produced by a straightforward approach.

1. INTRODUCTION

A main objective of structural testing is to achieve full or at least high structural coverage, such as statement coverage [2] and branch coverage [11] of the program under test. Passing tests that achieve high structural coverage not only indicate the thoroughness of the testing but also give high confidence of the quality of the program under test. Dynamic Symbolic Execution (DSE) [6, 9, 15] is a state-of-art technique, which can be employed to automatically generate test inputs that achieve high structural coverage. DSE instruments the program and executes the program with symbolic values as inputs. During the execution, DSE collects the path condition, which is the constraints on inputs obtained from the predicates in branch statements, and negates part of the constraints in the path condition to obtain a new path for further exploration.

When DSE is applied on real-world applications, DSE faces challenges in generating test inputs that can achieve high structural coverage. Among various issues related to these challenges, our preliminary study of applying DSE on open source projects shows that many blocks or branches in the program under test are not covered due to two main types of issues: (1) *object-creation issue (OCI)* and (2) *external-method-call issue (EMCI)*.

Object-Creation issue (OCI). To achieve high structural coverage of object-oriented program, the generated tests need to include specific method-call sequences for producing desirable object states of the receiver or arguments of the method under test. DSE typically identifies constructors and public setter methods for different fields of a class under test and uses them to form method-call sequences for producing desirable object states. An OCI occurs when DSE fails to generate method-call sequences to produce desirable object states for achieving new structural coverage. Figure 1 shows an example of OCI faced by DSE. For the method `TestPop`, DSE cannot create an object of the class `Stack`, whose size is more than 1, to cover the false branch of Line 6. The reason is that the method `Stack.Push` needs to be invoked to modify the value of the field `Stack.items`, but `Stack.Push` is not included in the method-call sequences generated for `Stack`, since `Stack.Push` is not a public setter method for any field of `Stack`.

External-Method-Call issue (EMCI). For the program under test, DSE does not by default instrument all third-party libraries since it would substantially increase the exploration space. Furthermore, DSE cannot instrument native system libraries that interact with environments, such as the file system and network transmission. Thus, if symbolic values are passed as arguments to methods that are not instrumented (i.e., methods in libraries that are not instrumented), DSE can have problems in achieving high coverage in two main situations: (1) Since DSE cannot structurally explore the code that is not instrumented, DSE cannot collect symbolic constraints resulted from executing external

```
public class Stack {
00:  private List<object> items = new List<object>();
01:  public void Push(object item) {
02:    items.Add(item); }
03:  public int Count() {
04:    return items.Count; }
05:  public object Pop() {
06:    if(items.Count() == 0) {
07:      throw new Exception("empty"); }
08:    object result = items[items.Count()-1];
09:    items.Remove(items.Count()-1);
10:    return result;}
11:   ...
12:  public void TestPop(Stack stack) {
13:    object item = stack.pop();
14:    ... }
```

Figure 1: Example of object-creation issue

methods and the return values of them are always concrete values. If such return values are used in deciding a subsequent branch, DSE may not collect the corresponding symbolic constraints for the path condition or cannot solve its new derived path condition correctly due to the missing symbolic constraints inside the external methods. In Figure 2, since the branch statement at Line 3 use the return value `ret` (a concrete value) of the external method `ExternalLib.Compute`, the constraint `ret > 5` cannot be collected. As a result, DSE cannot generate test inputs to satisfy the negation of this constraint needed for exploring the next new path. (2) If an external method throws an exception that aborts the program execution, and DSE cannot generate other inputs to cause the method not to throw an exception, DSE cannot explore the remaining parts of the program after the call site of this external method. In Figure 2, if the external method `ExternalLib.ThrowException` throws an exception when the symbolic value `z` is passed as an argument, the program execution is aborted. Since `ExternalLib.ThrowException` is an external method, DSE cannot collect the symbolic constraints that lead to the exception inside `ExternalLib.ThrowException`. If DSE cannot generate other values for `z` to cause `ExternalLib.ThrowException` not to throw an exception, DSE cannot explore the code from Line 7 to 9.

Automatically solving these two main types of issues is challenging. There exist several major method-call-sequence generation approaches based on bounded-exhaustive techniques [8, 21], evolutionary techniques [7, 19] , random techniques [4, 12], heuristic techniques [18], and mining production code techniques [17]. Nonetheless, these approaches cannot help DSE achieve branch coverage that is close to 100% [17], since the search space of method-call sequences is too huge. Similarly, instrumenting all third-party libraries can result in the explosion of exploration spaces, since DSE needs to explore many more paths inside the third-party libraries. By instrumenting these libraries, we may further get more external method calls since these libraries may invoke methods from other libraries that are not instrumented. Furthermore, native system libraries that interact with environments cannot be explored by DSE since they cannot be instrumented by DSE.

Although it is challenging to automatically solve these main types of issues, users can provide assistance to help DSE solve these issues. For example, to solve OCIs, users can provide factory methods that include method-call sequences to help DSE explore different object states. To solve EMCIs, users can specify which class or library to instrument, write mock objects, or set up the environments for the

```
// x, y, z are symbolic values of reference types
00:  string text = ExternalLib.Format(x);
01:  Console.Write("Result is: " + text);
02:  int ret = ExternalLib.Compute(y);
03:  if(ret > 5) {
04:    // do something
05:    ... }
06:  ExternalLib.ThrowException(z);
07:  if(z.GetValue() > 10) {
08:    // do something
09:    ... }
```

Figure 2: Example of external-method-call issue

external method calls that interact with the environments. One key insight is that, when provided with informative information of issues and the related not-covered statements or branches, users can effectively identify the problems and provide corresponding assistance to help DSE achieve high structural coverage.

A straightforward approach that reports every encountered issue can be employed to identify issues. For example, it can report every object type involved in a path condition that requires object states that cannot be produced by DSE, and report every encountered external method call during program executions. A major challenge of applying this approach is that many reported issues are false warnings, i.e., irrelevant to the problems that prevent DSE from achieving higher coverage. For example, it reports both `Stack` and `List<object>` as OCIs for the program shown in Figure 1, since the false branch of `items.Count() == 0` involves the object types of the argument `Stack` and its field `Stack.items`. However, the field `Stack.items` can be changed only by invoking the method `Stack.Push` or `Stack.Pop` in `Stack`, so reporting `List<object>` as an OCI is a false warning. Similarly, reporting the method `Console.Write` in Figure 2 as EMCI is also a false warning, since it only reads and prints the arguments. Such false warnings are referred as irrelevant issues in the following sections.

To address the major challenge in identifying issues faced by DSE, in this paper, we present a novel approach, called Covana. Covana effectively identifies issues that prevent DSE from achieving high coverage by analyzing runtime information, and filters out irrelevant issues using residual structural coverage [13], i.e., the structural coverage not yet achieved. In our approach, we first apply DSE on the program under test to generate test inputs and collect runtime information and residual structural coverage, such as not-covered statements or branches. Our approach then analyzes the runtime information to identify issues and uses the residual structural coverage to filter out irrelevant issues that are not related to not-covered statements or branches.

Furthermore, we propose two techniques that instantiate this general approach to effectively identify OCIs and EMCIs. To identify OCIs, our approach first extracts the object types of receiver or argument objects and their fields from path conditions. Our approach then analyzes whether the fields can be assigned by invoking a constructor or a public setter method of their declaring classes to identify the object types whose states cannot be produced by DSE.

To identify EMCIs, our approach assigns symbolic trackers[1] to the return values of the external method calls. In this way, our approach can identify external methods whose

[1]A symbolic tracker is a lightweight symbolic value that can be used to collect constraints in branch statements, but the collected constraints on the symbolic value are intentionally ignored by the constraint solver.

return values are used in one of the subsequent branches. Moreover, our approach monitors program executions to identify the external method calls that throw exceptions. All these identified issues are further filtered out by using the information of residual structural coverage.

This paper makes the following major contributions:

- We propose the first general approach to identify issues that prevent DSE from achieving high structural coverage by analyzing runtime information of DSE, and filter out irrelevant issues using residual structural coverage.
- We propose a technique that extracts the object types of receiver or argument objects and their fields from path conditions, and reports only the object types whose states cannot be produced by DSE as OCIs.
- We propose a novel technique that identifies two types of external methods as EMCIs: (1) external methods whose return values are used in one of the subsequent branches; (2) external methods that throw exceptions preventing DSE from exploring remaining parts of the program after the call sites of the external methods.
- We have implemented our approach as a tool upon a state-of-the-art industrial testing tool, Pex [18], and conducted evaluations on open source projects. Our results show that our general approach instantiated with two techniques effectively identifies 155 OCIs and 43 EMCIs. Moreover, our evaluation results show that our approach effectively reduces 296 irrelevant issues out of 451 OCIs and 1567 irrelevant issues out of 1610 EMCIs produced by a straightforward approach.

2. DYNAMIC SYMBOLIC EXECUTION

Dynamic Symbolic Execution (DSE) executes the program symbolically, starting with arbitrary inputs. Along the execution path, DSE collects symbolic constraints on program inputs in branch nodes (being runtime instance 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 to the node being negated with the old path. Then a constraint solver is used to compute test inputs that satisfy the new path condition. These generated test inputs again are fed into the program to explore different paths of the program. Ideally, all feasible paths can be exercised eventually through such iterations of path variations. However, when DSE is applied on real-world applications, various issues can occur to prevent DSE from achieving high structural coverage. Algorithm 2.1 [22] shows the general iterative DSE algorithm.

In practice, it turns out that choosing program input i to satisfy $\neg J(i)$ (new path condition resulted from branch-node negation) is not always possible for various reasons: (1) Some path conditions obtained by negating part of the constraints from path condition can be infeasible. (2) Some path conditions cannot be solved due to the limitation of the used constraint solver (e.g. floating-point arithmetics). (3) Some path conditions are too complex and require too much time to solve. (4) Some path conditions may require object states that cannot be produced by the method-call sequences generated by DSE, corresponding to OCIs.

Algorithm 2.1 Dynamic Symbolic Execution (DSE)

```
/*J is the set of already analyzed program inputs*/
Set J := φ;
loop
  Choose program input i such that ¬J(i)
  stop if no such i can be found
  Output i;
  Execute P(i); record path condition C /*C(i) holds*/
  Set J := J ∨ C /*viewing C as the set {i|C(i)}*/
end loop
```

After DSE generates new test inputs, DSE instruments the program and executes the program with these inputs. By default, DSE does not instrument all third party libraries due to explosion of the exploration spaces, and DSE cannot instrument native system libraries that interact with environments, such as file system and network transmission libraries. Thus, symbolic values may flow to methods that are not instrumented. In this case, DSE may not achieve high structural coverage if the return values of the external method calls are used in subsequent branch statements or if exceptions are thrown during the execution of the external method calls to abort the whole program executions.

Boundary issues are issues that occur during program executions, mainly caused by loops or too many paths in the program under test. The program under test may have loops whose number of iterations depend on the symbolic values or whose bodies include branches executed for many times or infinitely. DSE iteratively explores the paths inside loops, preventing DSE from exploring new paths in the remaining parts of the program. To prevent such issues in keeping DSE busy in negating the branch nodes inside loops, DSE provides default boundary values for imposing bounds for DSE, e.g., the maximum number of exploration iterations and the maximum number of negated branch nodes in a path.

To inform users the issues that prevent DSE from achieving high structural coverage, DSE needs to report these issues with the related not-covered statements or branches. Our Covana approach, described in section 5, can effectively identify these issues, with OCIs and EMCIs as our focus, and reduce irrelevant issues reported by a straightforward approach.

3. PRELIMINARY STUDY

In this section, we discuss the distribution of the issues that we empirically observed by applying a state-of-the-art DSE tool, Pex [18], on open source projects for achieving high structural coverage. The analysis of these issues helps motivate our approach. We choose Pex as the DSE tool in our empirical study and later we implemented 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 issues [18]. We apply Pex on the core libraries of SvnBridge [16], xUnit [23], Math.NET [10], and QuickGraph [14], until all the methods has been explored by Pex or Pex run out of memory and cannot continue to generate test inputs. After Pex generates test inputs and outputs coverage files, we select 10 source files that achieve low coverage in each project, and manu-

Project	Cov %	OCI	EMCI	Boundary	Limitation
SvnBridge	56.26	11 (42.31%)	15 (57.69%)	0 (0%)	0 (0%)
xUnit	15.54	8 (72.73%)	3 (27.27%)	0 (0%)	0 (0%)
Math.Net	62.84	17 (70.83%)	1 (4.17%)	4 (16.67%)	2 (8.33%)
QuickGraph	53.21	10 (100%)	0 (0%)	0 (0%)	0 (0%)
Total	49.87	46 (64.79%)	19 (26.76%)	4 (5.63%)	2 (2.82%)

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

ally investigate the issues that contribute to the not-covered statements and branches.

Table 1 shows the distribution of the issues that prevents DSE from achieving high structural coverage. Column "Project" lists the name of each project and Column "Cov %" shows the block coverage achieved by Pex. The other four columns gives the number and the percentage of the not-covered branches caused by different types of issues.

The top main type of issues is object-creation issues (64.79%), shown in Column "OCI", since DSE cannot generate desirable objects required to cover certain branches. Among the programs that have OCIs, we observed that many generated test inputs contain method-call sequences to produce not null argument objects with `null` object for their fields, since DSE cannot figure out method-call sequneces to produce desirable objects for the fields. In this case, if we use a straightforward approach to report all the object types involved in these path conditions, we can get OCIs of the argument objects and their fields, resulting in the irrelevant issues of the argument objects.

The second main type of issues is external-method-call issue (26.76%), shown in Column "EMCI", since symbolic values flow to methods that are not instrumented. In our study, we encountered many external method calls, 405 in 40 files, but only 4.7% (19 in 405) caused problems to prevent DSE from achieving high structural coverage. If we use a straightforward approach to report all these external methods, we can get many irrelevant issues that are not related to any not-covered statment or branch.

The third main type of issues is boundary issues (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 negating branch nodes to increase the number of iterations of the loops, which prevents DSE from exploring other paths in the remaining parts of the program.

The last main type of issues 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 issues is that DSE generates approximate integers for constraints that contain floating-point arithmetics, which are able to cover certain branches.

Our approach proposed in this paper is a general approach to effectively identify issues that prevent DSE from achieving high structural coverage. Since the top two main types of issues are OCIs and EMCIs, which cannot be effectively identified by the straightforward approach, we propose two specific techniques to instantiate our general approach to effectively identify these two main types of issues. In addition, our approach can effectively reduce irrelevant issues produced by the straightforward approach.

4. EXAMPLE

We next explain how our approach, instantiated with two specific techniques, identifies OCIs and EMCIs with two illustrative examples.

4.1 Object-Creation Issue (OCI)

Figure 3 shows a class `FixedSizeStack` that has a field `stack` of type `Stack`, which is shown in Figure 1. `FixedSizeStack` has a upper bound of the number of objects that can be pushed into the stack. To guarantee the size bounding, the method `FixedSizeStack.Push` throws an exception when the size of a stack has reached the bound (10 in the example). The method `TestPush` receives an object as argument and invokes the method `FixedSizeStack.Push` to push the object to the stack for testing. To cover the true branch at Line 4 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, the difficulty of generating an object of `FixedSizeStack` whose size is 10 lies in generating an object of `Stack` whose size is equal to 10. However, as discussed in Section 1, DSE cannot generate an object of `Stack` whose size is more than 1, since its field `Stack.items` can be changed only by invoking the method `Stack.Push`. In this case, we need to report an OCI of the class type `Stack`.

To identify this OCI, our approach first collects path conditions for which DSE cannot produce desirable object states. In this example, the collected path condition is the path condition that leads to a path follows the true branch at Line 4. From this path condition, our approach extracts the involved object types of the argument object `FixedSizeStack` and its fields, `FixedSizeStack`, `FixedSizeStack.stack`, and `Stack.items`. By analyzing the argument object `FixedSizeStack` and its fields, our approach identifies `Stack` as the object type that causes an OCI and reports it as a candidate issue. Since the path condition results in a new path that follows the true branch at Line 4, which is not covered, our approach confirms this candidate issue as an OCI related to a not-covered branch.

4.2 External-Method-Call Issue (EMCI)

EMCIs occur when symbolic values flow to methods that are not instrumented and whose return values are used to

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

Figure 3: FixedSizeStack implemented using Stack

decide a subsequent branch. In the example shown in Figure 2, both `ExternalLib.Format` and `ExternalLib.Compute` receive symbolic values `x` and `y` as arugments, and have return values, `text` and `ret`, respectively. By tracking `text`, our approach detects that `text` is passed to `Console.Write` as an argument. Since `Console.Write` does not has any return value and only reads the value from `text`, `Console.Write` does not cause any problem for DSE. After `text` flows to `Console.Write`, `text` is never used in any subsequent branch statement, so `ExternalLib.Format` is not considered as an issue. By tracking `ret`, our approach detects that `ret` is used in the branch statement at Line 3, and reports `ExternalLib.Compute` as a candidate EMCI. Our approach then checks the residual structural coverage to see whether the branches at Line 3 is covered. If one of the branches is not covered, `ExternalLib.Compute` is reported as an EMCI related to the not-covered branch at Line 3.

Another situation where EMCIs occur is external methods that receive symbolic values as arguments throw exceptions to abort program executions, preventing DSE from exploring the remaining parts of the program after the call sites of external methods. In the example, the method `ExternalLib.ThrowException` receives a symbolic value, `z`, as an argument. Instead of returning a value, `ExternalLib.ThrowException` throws an exception. If DSE cannot generate different values of `z` that can cause `ExternalLib.ThrowException` not to throw an exception, the remaining parts of program after the call site of `ExternalLib.ThrowException`, Lines 7 to 9, cannot be explored or covered by DSE. To detect such situation, our approach collects runtime informtion about external methods that throw exceptions. In our example, our approach identifies `ExternalLib.ThrowException` as a candidate EMCI by extracting external methods that throw exceptions from the collected runtime information. By checking the residual structural coverage of Lines 7 to 9, if none of them is covered, our approach confirms this external method call as an EMCI related to Lines 7 to 9.

5. APPROACH

Figure 4 shows the overview of our Covana approach. Our approach consists of three main components: information collector, issue analyzer, and irrelevant issue filter. The information collector is integrated into a DSE engine to collect runtime information, such as path conditions and external method calls, and residual structural coverage, such as not-covered statements and branches. The issue analyzer accepts the runtime information and identifies different types of issues. The irrelevant issue filter accepts the candidate issues produced by the issue analyzer and uses the residual structural coverage to filter out candidate issues that are not related to any not-covered statements or branches. In our current prototype, we instantiate this general approach with two techniques to identify the top two main types of issues: OCIs and EMCIs. We next discuss these three main components with the specific information collectors, issue analyzers, and filters for identifying OCIs and EMCIs.

5.1 Information Collector

The information collector is integrated into the DSE engine to collect necessary runtime information for issue analysis. The runtime information of DSE includes the path conditions used to generate test inputs, the external method call encountered, the boundaries exceeded (e.g., timeout and too many exploration iterations) and so on. In our current prototype, to collect the information of path conditions and external method calls for analyses of OCIs and EMCIs, we provide three information collectors: *path-condition collector*, *method-result tracker*, and *exception collector*.

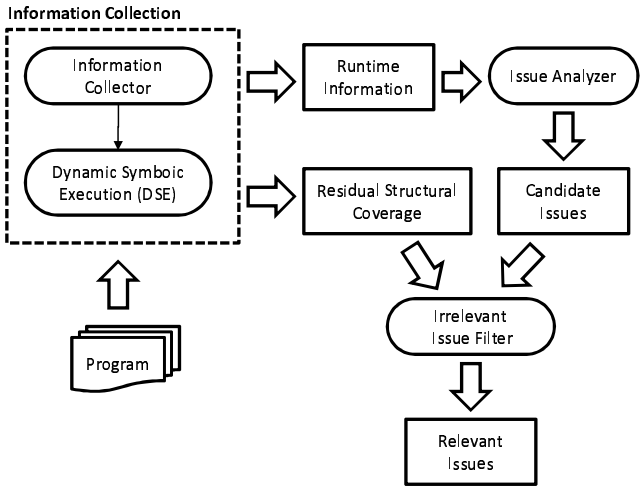


Figure 4: Overview of Covana

5.1.1 Path-Condition Collector

After each execution of the program under test, DSE strategically selects a branch node[2] of the collected path condition and negate the branch node to obtain a new path condition. This new path condition is sent to a constraint solver for computing new test inputs that follow a new path. If the new path condition requires object states that cannot be produced by DSE, the path-condition collector records the path condition and the related branch whose branch node is negated. In the program shown in Figure 3, the collected path condition is `FixedSizeStack s3 = new` $\wedge$ `Stack s2 = s3.stack` $\wedge$ `List<object> s1 = s2.items` $\wedge$ `int s0 = s1._size` $\wedge$ `s0 == 10`, and the branch collected is the true branch at Line 4.

5.1.2 Method-Result Tracker

The method-result tracker receives a callback from the DSE engine when the execution of a method call is finished. If the method is not instrumented by DSE, the method call is an external method call and each argument of the method is checked to see whether the argument is a symbolic value or not. If any argument is a symbolic value, our approach creates a *symbolic tracker* and assign it to the return value of the external method call. The symbolic values of these symbolic trackers carry the information of the external methods and can be used by the DSE engine to collect symbolic constraints on these symbolic trackers. However, the collected constraints on the symbolic trackers are intentionally ignored by the constraint solver when computing new inputs. As a result, while reducing the performance overhead, our approach does not have any side effect to the original executions during information collection. These symbolic trackers result in the constraints on the return values in the subsequent branch statements to be collected into the path condition. The method-result tracker records the path condition and its related branches when the method-result tracker identifies any external methods to track. In the code

[2]A branch node represents an runtime instance of a conditional branch in the code during execution.

snippet shown in Figure 2, our approach tracks the external methods `ExternalLib.Format` and `ExternalLib.Compute`, since both of them receive symbolic values as arguments. The external method `Console.Write` receives the return value `text` of `ExternalLib.Format` as part of the argument, but it does not have any return value. Thus our approach simply ignores `Console.Write`. Since `text` is not used in any subsequent branch statement, no symbolic constraints on `text` is collected into the path condition. In this way, `ExternalLib.Format` is filtered out automatically. The return value `ret` of `ExternalLib.Compute` is used in the branch statement at Line 3, so the symbolic constraints on `ret` is collected into the path condition for later issue analysis. The related branches of this path condition are the branches at Line 3.

5.1.3 Exception Collector

Whenever an exception is thrown during program execution, the exception collector receives a callback from DSE; this callback provides the information of the method that throws the exception, the input arguments of the method, and the exception. If the method is an external method and the input arguments contain symbolic values, the exception collector collects the information of the method and the exception. In the code snippet shown in Figure 2, the method `ExternalLib.ThrowException` is collected for later issue analysis, since `ExternalLib.ThrowException` throws an exception when receiving the symbolic value `z` as the argument.

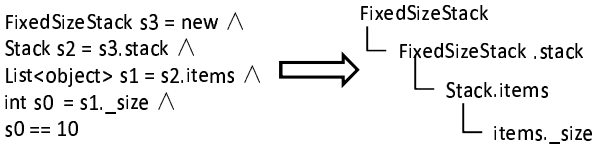
5.2 Issue Analyzer

The issue analyzer takes runtime information of DSE as input and identifies different types of candidate issues that may prevent DSE from achieving high structural coverage. In our current prototype, we provide object-creation-issue analyzer and external-method-call-issue analyzer to identify OCIs and EMCIs.

5.2.1 Object-Creation-Issue Analyzer

The object-creation-issue-analyzer accepts the path conditions collected by the path-condition collector to identify OCIs. From the symbolic values in a path condition, our approach extracts the object types of receiver or argument objects and their fields. In the example shown in Figure 3, our approach extracts the object type of the argument object, `FixedSizeStack`, and its fields `FixedSizeStack.stack`, `Stack.items`, and `List<object>._size` from the path condition collected by the path-condition collector.

If our approach cannot extract any field from the path condition, our approach directly report the object type of the receiver or argument object as an OCI. In our example of `FixedSizeStack`, since there are three fields extracted from the path condition, our approach need further analysis of these fields to identify the object type whose state cannot be produced by DSE. The object type of argument object and its fields extraced from the path condition are shown as below.



If a field can be assigned by invoking a constructor or a public setter method of its declaring class, DSE can 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, not its declaring type, that causes an OCI. In our example, `FixedSizeStack.stack` can be assigned by invoking the construcor of `FixedSizeStack`. Thus, to create an object of `FixedSizeStack` whose size is 10, DSE needs to create an object of `Stack` and assign it to `FixedSizeStack.stack` by passing it as an argument to the construcor of `FixedSizeStack`.

```
Issue ObjectCreationIssueAnalysis(PathCondition pc) {
00:  TypeEx argType = pc.GetArgumentObjectType();
01:  Field[] fields = pc.GetInvolvedField();
02:  Branch branch = pc.GetRelatedBranch();
03:  if(fields.Length == 0) {
04:    return new ObjectIssue(argType,branch)); }
05:  Field current = null;
06:  for(int i = 0;i < fields.Length;i++) {
07:    current = fields[i];
08:    TypeEx dec = current.GetDeclaringType();
09:    if(!dec.HasConstructorFor(current) &&
          !dec.HasSetterMethodFor(current)) {
10:      return dec; } }
11:  return new ObjectIssue(current.GetType(),branch); }
```

Figure 5: Object-Creation Issue Analysis

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 methods of its declaring class. Hence, its declaring class type should be reported as an OCI. In our example, the field `Stack.items` cannot be assigned with an object of `List<object>` by invoking any constructor or public setter method of `Stack`. Thus, to change the object state of `Stack.items`, DSE need specific method-call sequences of `Stack` instead of `List<object>`. As a result, our approach reports the object type `Stack` as a candidate OCI. The pseudocode for identifying the object type is shown in Figure 5.

5.2.2 External-Method-Call-Issue Analyzer

The external-method-call-issue analyzer accepts the path conditions collected by the method-result tracker and the exceptions and methods collected by the exception collector to identify EMCIs. To identify external methods whose return values are used in subsequent branch statements, the external-method-call-issue analyzer extracts the symbolic trackers from the path conditions. From these symbolic trackers, the analyzer extracts the information of the corresponding external methods. For the code snippet shown in Figure 2, the external method extracted is `ExternalLib.Compute`. The external-method-call-issue analyzer then constructs a candidate EMCI using the information of the extracted external method (`ExternalLib.Compute`) and the related branches of the path condition (the branches at Line 3). To identify external methods that throw exceptions when receiving symbolic values as arguments, the analyzer first extracts the call site of the external methods from the exception information. In our example, the call site is the statement at Line 6, which invokes the external method `ExternalLib.ThrowException`. The analyzer then uses the call site and the recored external method, `ExternalLib.ThrowException`, to create a candidate EMCI.

5.3 Irrelevant Issue Filter

The irrelevant issue filter uses the residual structural coverage to filter out the candidate issues that are not related to any not-covered statements or branches. We provide irrele-

vant issue filters for OCI and EMCI in our current prototype.

5.3.1 Irrelevant-Object-Creation-Issue Filter

For a candidate OCI produced from a path condition, our approach checks whether the related branch of the path condition is covered. If it is included in the residual structural coverage, our approach figures out that the branch is not covered. Otherwise, our approach filters out the candidate issue. In the example shown in Figure 3, the related branch of the path condition, from which the OCI of `Stack` is reported, is the true branch at Line 4. If this branch is not covered, our approach does not filter out the OCI of `Stack`. The output of the filter is the OCIs that are related to not-covered branches.

5.3.2 Irrelevant-External-Method-Call-Issue Filter

For the candidate EMCIs whose return values are used in subsequent branches, our approach uses the residual structural coverage to check whether one of these branches is not covered. If all these branches are covered, our approach filters out the candidate issues. In the code snippet shown in Figure 2, if both branches at Line 3 are covered, the candidate EMCI of `ExternalLib.Compute` is filtered out. For the candidate EMCIs that throws exceptions, our approach uses the residual structural coverage to check whether the remaining parts of the program, i.e., the parts that can be executed after the normal return of the external method, are covered. If the remaining parts of the program are already covered by previously generated test inputs, our approach filters out the candidate issue. In the code snippet shown in Figure 2, the remaining parts our approach need to check are Lines 7 to 9 in Figure 2, since the exception is thrown from the statement at Line 6 and there is no branches after Line 6. If Lines 7 to 9 are already covered, our approach filters out the candidate EMCI of `ExternalLib.ThrowException`.

6. EVALUATIONS

We conducted two evaluations to show the effectiveness of Covana. In our evaluations, we used two popular .NET applications: xUnit [23] and QuickGraph [14]. In our evaluations, we answer the following research questions:

- **RQ1**: How effective is Covana in identifying the two main types of issues, OCIs and EMCIs?
- **RQ2**: How effective is Covana in reducing the issues reported by the straightforward approach?

We next provide details of the metrics that we collect in our evaluations. To measure the effectiveness of our approach in identifying OCIs and EMCIs, we measure the number of issues that are found for the not-covered branches of the subject applications. We then measure the false positives, i.e., the number of irrelevant issues that are identified by our approach as relevant issues and the false negatives, i.e., the number of relevant issues that are not identified as relevant issues. To measure the effectiveness of our approach in reducing irrelevant issues produced by the straightforward approach [3], we compare the number of issues identified by the straightforward approach with the number of issues identified by our approach in applications under test and measure the number of issues reduced by our approach. In addition, we measure the false positives, i.e., the real irrelevant issues reported by the straightforward approach but not identified by Covana, and the false negatives, i.e., the real relevant issues reported by the straightforward approach but identified as irrelevant issues by Covana.

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

6.1 Subjects and Evaluation Setup

We used two popular .NET applications for evaluating our Covana approach: xUnit [23] and QuickGraph [14]. xUnit is a unit testing framework for .NET program development. xUnit includes 223 classes and interfaces with 13 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 [3]. QuickGraph includes 165 classes and interfaces with 5 KLOC.

In our evaluations, we use Pex [5, 18] as our DSE test generation tool. We implemented the information collectors of our approach as extensions to Pex for the information collection. We first applied Pex on each subject application to generate test inputs and collect runtime information and residual structural coverage. Using the collected information as input, the analysis tool implemented based on our approach is executed to identify OCIs and EMCIs. To compare the effectiveness of our approach with the straightforward approach, we also implemented an analysis based on the straightforward approach and executed it with the collected information to identify OCIs and EMCIs.

We next discuss the results of our evaluations in terms of the effectiveness of Covana in identifying OCIs and EMCIs, and in reducing the irrelevant issues reported by the straightforward approach.

6.2 RQ1: Effectiveness

In this section, we address the research question RQ1 of how effectively Covana identifies OCIs and EMCIs. To address this question, we measure the number of false positives and the number of false negatives generated by Covana. To measure values for these metrics, we executed the analysis tool implemented using our approach with the information collected from Pex as inputs, and manually classified the issues reported by the tool as false positives and false negatives.

Table 2 shows the results for all the assemblies in both subject applications. Columns "Application Assembly" and "# File" list the number of source files in each application assembly. Columns "Object-Creation Issue" and "External-Method-Call Issue" show the OCIs and EMCIs identified by Covana. Subcolumn "# Issue" gives the number of relevant issue identified by us manually. Subcolumn "Found" gives the number of relevant issues 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 155 OCIs with 20 as false positives and 28 as false negatives. The reason why we have 17.17 % (28 in 163) issues as 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 addition, our approach identifies 43 EMCIs with only 1 false positive and 2 false negatives. We

[3]The straightforward approach reports every object type of argument object and its fields in the path condition and reports every encountered external method calls

Application Assembly	# File	Object-Creation Issue				External-Method-Call issue			
		# Issue	# Found	# FP	# FN	# Issue	# Found	# FP	# FN
xUnit	71	67	68	13	11	24	24	0	0
xUnit.Extensions	17	5	7	3	1	2	2	0	0
xUnit.Console	7	2	2	0	0	2	2	0	0
xUnit.Gui	12	3	3	0	0	3	1	0	2
xUnit.Runner.Msbuild	6	14	15	1	0	0	0	0	0
xUnit.Runner.Tdnet	3	5	5	0	0	1	1	0	0
xUnit.Runner.Utility	28	12	7	0	4	9	9	0	0
Quickgraph	3	0	0	0	0	0	0	0	0
Quickgraph.Algorithms	12	11	7	0	4	0	0	0	0
Quickgraph.Algorithms.Graphviz	14	20	20	2	2	3	4	1	0
Quickgraph.Collections	19	11	6	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	163	155	20	28	44	43	1	2

Table 2: Evaluation results showing the effectiveness of Covana in identifying OCIs and EMCIs

```
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:  public Exception ClassFinish() {
12:    foreach (object fixtureData in fixtures.Values) {
13:      ... } } }
```

Figure 6: `TestClassCommand` class of xUnit

discuss the details of the false negative that we found for EMCIs in Section 7.

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

Figure 6 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%). The generated test input included method-call sequences that invoked the constructor of `TestClassCommand` and passed `null` as the argument to create an object of `TestClassCommand`. This generated test input indicates that Pex has figured out how to create objects of `TestClassCommand`. In the method `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 OCI of the interface type `ITypeInfo`. By analyzing path conditions that involve private fields of objects that are test inputs, 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 `TestClassCommand`, and reports an OCI 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 12 requires the field `TestClassCommand.fixtures` to hold at least one item. Since there is no constructor or public setter method to assign an object to `TestClassCommand.fixtures`, other public methods of `TestClassCommand` need to be invoked to change the value of `TestClassCommand.fixtures`. Thus, we need to report `TestClassCommand` as an OCI. 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 OCI for the not-covered branch at Line 12.

Figure 7 shows two methods: (1) the method `ParseCommandLine` of the 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]` (a symbolic value), our approach assigned a symbolic tracker to the return value of `File.Exists` and found that the return value was used in the not-covered branch at Line 1 (the false branch). Thus, our approach correctly reported an EMCI 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 and `Path.GetFullPath` during information collection. 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 EMCI. Although another external method `Console.WriteLine` at Line 2 received `assemblyFile` as part of the argument, this external method did not have any return value. As a result, our approach correctly filtered out this external method `Console.WriteLine`.

6.3 RQ2: Issue Reduction

In this section, we address the research question RQ2 of how effectively our approach reduces the number of issues reported by the straightforward approach. To address this

Application	Object-Creation Issue					External-Method-Call Issue				
	# SF	# Covana	# Reduced	# FP	# FN	# SF	# Covana	# Reduced	# FP	# FN
xUnit	335	107	228 (68.06%)	17	16	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	28	1610	43	1567(97.33%)	1	2

Table 3: Evaluation results showing the effectiveness of Covana in reducing issues produced by the straightforward approach

```
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; }
...
public Executor(string assemblyFilename) {
04:  this.assemblyFilename = Path.GetFullPath(assemblyFilename);
05:  ... }
```

Figure 7: Two methods that have EMCIs in xUnit

question, we compare the number of issues reported by the straightforward approach and our approach, and measure the number of issues reduced by our approach. In addition, we measure the false positives, i.e., the real irrelevant issues reported by the straightforward approach but not identified by Covana, and the false negatives, i.e., the real relevant issues reported by the straightforward approach but identified as irrelevant issues by Covana. To measure values for these metrics, we executed the analysis tool implemented based on the straightforward approach with the information collected from Pex as inputs, and manually classified the issues reduced by our tool as false positives and false negatives.

Table 3 shows the results of both subject applications. Column "Application" lists the names of the subject applications. Columns "Object-Creation Issue" and "External-Method-Call Issue" show the OCIs and EMCIs identified by the straightforward approach and Covana, respectively. Subcolumn "# SF" gives the number of issues identified by the straightforward approach, subcolumn "Covana" gives the number of issues identified by Covana, and subcolumns "# FP" and "# FN" give the number of false positives and false negatives. The results show that our approach reduces 65.63% (296 in 451) OCIs with 20 as false positives and 28 as false negatives, and reduces 97.33% (1567 in 1610) EMCIs with only 1 false positive and 2 false negatvies. These results show that our approach effectively reduces the irrelevant issues produced by the straighforward approach with low false positives and false negatives.

7. DISCUSSION AND FUTURE WORK

In this section, we discuss some of the issues of the current implementation of our Covana approach and how they can be addressed.

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 trackers to these static fields, so that our approach can collect the symbolic constraints on these static fields for our analysis.

```
// methods in Line 1, 2, 4 are external methods
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:      if (itemKey != null) {
08:        ... } } } }
```

Figure 8: The method `LoadAssemblyList` in the class `RecentlyUsedAssemblyList` of xUnit

Precise Analysis of Path Condition. In our current implementation, we extract object types of argument objects and their fields from a path condition. In this way, if there is only one solution to a path condition, our approach works correctly. However, some path conditions contain disjunctions of complex conditions, which can result in several possible solutions to the path condition. In this case, our approach needs to check every possible solution to the path condition and find out why the path condition cannot be satisfied. We plan to integrate a more precise path condition analyzer to our approach in future work, so that our approach can carry out the analysis more effectively.

Concrete Arguments for External Method Call. Our current implementation tracks the return values of external methods if the methods receive symbolic values. However, in our evaluation, there were a few cases where the external methods received concrete values as arguments, and resulted in some not-covered branches. The external method `recentKey.OpenSubKey`, shown in Figure 8, 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 track the return value of any external method that does not receive any symbolic values as an argument. Using symbolic trackers, our approach can be easily extended to track every external method call, no matter whether it receives symbolic values as arguments or not. However, tracking every external method call may incur many false positives and increase the performance overhead significantly, since the number of encountered external method calls during the program executions is not trivial. In future work, we plan to conduct experiments to measure the effectiveness and performance overhead when our approach tracks every external method call.

8. RELATED WORK

Coverage Analysis. Pavlopoulou and Young [13] 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 test coverage from deployed software, their approach did not provide a way to analyze the coverage, while our approach analyzes the residual structural coverage gathered from DSE to filter out irrelevant issues.

Explaining Failures of Program Analysis. Dincklage and Diwan [20] propose an analysis language and build a system to produce reasons when the 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 issues, our approach focuses on a significantly different problem and includes significantly different techniques needed for addressing unique challenges in identifying and filtering issues that prevent DSE 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 application 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 EMCIs, our approach not only checks whether external methods receive symbolic values as arguments, but also tracks their return values to determine whether their return values are used in not-covered branches.

9. CONCLUSION

The process of achieving high structural coverage of the program under test can be automated using DSE, which generates test inputs to iteratively explore paths of the program under test. When DSE is applied on real-world applications, there are a number of issues that prevent DSE from achieving high structural coverage, with OCIs and EMCIs as the top two main types of issues. To effectively identify these issues for users to provide assistance to DSE, we presented Covana, a general approach to identify issues by analyzing runtime information and filter out irrelevant issues by using the residual structural coverage of DSE. We provided two techniques to instantiate our general approach to identify OCIs and EMCIs by analyzing path conditions and external method calls collected during the test generation of DSE. We conducted evaluations on two open source projects to demonstrate the effectiveness of Covana. The results show that Covana effectively identifies 155 OCIs, and 43 EMCIs. In addition, Covana reduces irrelevant issues including 65.63% (296 in 451) OCIs and 97.33% (1567 in 1610) EMCIs produced by the straightforward approach with low false positives and false negatives.

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] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., 1990.

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

[4] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Softw., Pract. Exper.*, 34(11):1025–1050, 2004.

[5] J. de Halleux and N. Tillmann. Parameterized Unit Testing with Pex. In *Proc. TAP*, pages 171–181, 2008.

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

[7] K. Inkumsah and T. Xie. Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution. In *Proc. ASE*, pages 297–306, 2008.

[8] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proc. TACAS*, pages 553–568, 2003.

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

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

[11] S. C. Ntafos. A Comparison of Some Structural Testing Strategies. *IEEE Trans. Softw. Eng.*, 14(6):868–874, 1988.

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

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

[14] QuickGraph: A 100% C# graph library with Graphviz Support, 2008. `http://www.codeproject.com/KB/miscctrl/quickgraph.aspx`.

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

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

[17] 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.

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

[19] P. Tonella. Evolutionary testing of classes. In *Proc. ISSTA*, pages 119–128, 2004.

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

[21] T. Xie, D. Marinov, and D. Notkin. Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests. In *Proc. ASE*, pages 196–205, 2004.

[22] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-Guided Path Exploration in Dynamic Symbolic Execution. In *Proc. DSN*, June-July 2009.

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