Covana: Precise Identification of Problems in Pex

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

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

Categories and Subject Descriptors

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

General Terms

Measurement, Reliability

Keywords

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

1. INTRODUCTION

Achieving high structural coverage (e.g., statement coverage, block coverage and branch coverage) is an important goal of software testing. However, manually producing high-covering 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 test-generation approaches to automatically generate test inputs, such as Pex [5] built based on Dynamic Symbolic Execution (DSE) [3, 4] (also called concolic testing [4]).

Although 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. Our preliminary study [8] shows that 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 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, we propose a new methodology of cooperative developer testing, where developers provide corresponding guidance to help the tools address the problems. One challenge in this methodology is that the tools need to report the encountered problems and narrow down the investigation scope, thus reducing the required efforts from the developers.

To explore the methodology of cooperative developer testing, our research centers around Pex [5], a state-of-the-art automated test-generation tool built for DSE. Pex takes as inputs the program under test or Parameterized Unit Tests (PUT) [6], i.e., unit test methods with parameters. Pex explores the program under test symbolically, and produces test inputs for the program under test or PUTs. Pex also outputs the achieved block coverage in the format of HTML reports. When applied on complex programs in practice, Pex cannot easily achieve high structural coverage mainly due to OCPs and EMCPs. To help Pex achieve higher structural coverage, developers can provide guidance to help Pex solve the problems. For example, to deal with OCPs, developers can specify factory classes [5] that encode desired method sequences for non-primitive object types. To deal with EMCPs, developers can configure Pex to instrument the external-method calls or provide mock objects [7] to simulate environment dependencies. By taking these guidances, Pex can be re-applied to achieve higher coverage.

To achieve this cooperative developer testing, Pex needs to inform the developers of what problems prevent Pex from achieving high structural coverage. Indeed, Pex reports various types of encountered problems during test generation

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA
ACM 978-1-4503-0445-0/11/05.

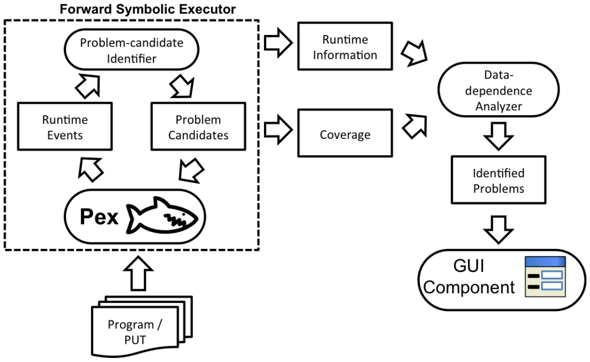


Figure 1: Architecture of Covana

and execution. For example, EMCPs are reported as problems of uninstrumented methods and external methods [5] and OCPs are reported as object creation problems. However, many of the reported problems are not the causes for Pex not to achieve high coverage. Such problems are referred to as irrelevant problem candidates. For example, Pex reports an EMCP when an external method is invoked during test execution, resulting in many false warnings. Pex reports OCPs when it fails to generate desired object states of program inputs to satisfy some path conditions. However, Pex does not analyze which fields of the program inputs require desired object states. These irrelevant problem candidates require extra efforts from the developers to identify the real problems that prevent Pex from achieving high coverage.

To address the challenge of reporting problems that prevent Pex from achieving high coverage and reducing the efforts of developers in providing guidance to Pex, we developed Covana[1], a tool that precisely identifies the problems (with the focus on EMCPs and OCPs) that prevent automated test-generation tools from achieving high structural coverage. Our Covana tool identifies problems by dynamically computing data dependencies of branch statements containing not-covered branches (referred to as partially-covered branch statements) and pruning problem candidates (referred to as irrelevant ones) that partially-covered branch statements have no data dependencies on. To identify EMCPs, our Covana tool identifies as problem candidates the invoked external methods and prunes irrelevant external-method calls by using the exception information and the data dependencies on external-method calls for their return values. To identify OCPs, Covana computes data dependencies of partially-covered branch statements on program inputs and their fields, and analyzes the field declaration hierarchy constructed from the fields that partially-covered branch statements have data dependencies on up to program inputs to identify the fields that require desired object states to cover certain not-covered branches.

2. COVANA ARCHITECTURE

Covana consists of three main steps: (1) identifying problem candidates using runtime information; (2) assigning symbolic values to elements of problem candidates (such as return values of external-method calls) and performing forward symbolic execution [5] using test inputs generated by the tools as program inputs; (3) pruning the irrelevant problem candidates that none of partially-covered branch statements have data dependencies on. Figure 1 shows the high-level overview of Covana's architecture. The architecture consists of three major components: the forward symbolic executor, the data-dependence analyzer, and the Graphic User Interface (GUI) component. The forward symbolic executor takes as inputs a program or Parameterized Unit Tests (PUT) [2] and produces test inputs, the achieved coverage information, and the runtime information (e.g., symbolic expressions of predicates in branch statements and exceptions). The data-dependence analyzer takes the coverage information and the runtime information as inputs and produces the identified problems, which are the output of the tool. These identified problems are then shown by the GUI component. We next describe these components in detail. Currently, Covana supports the detection of EMCPs and OCPs only.

2.1 Forward Symbolic Executor

The forward symbolic executor component performs forward symbolic execution and produces the achieved coverage and the collected runtime information. Covana leverages the DSE engine of Pex to perform forward symbolic execution. To identify problem candidates and collect runtime information, we implemented the problem-candidate identifier as a Pex extension. The problem-candidate identifier observes the runtime events from Pex and analyzes these events to identify different types of problem candidates.

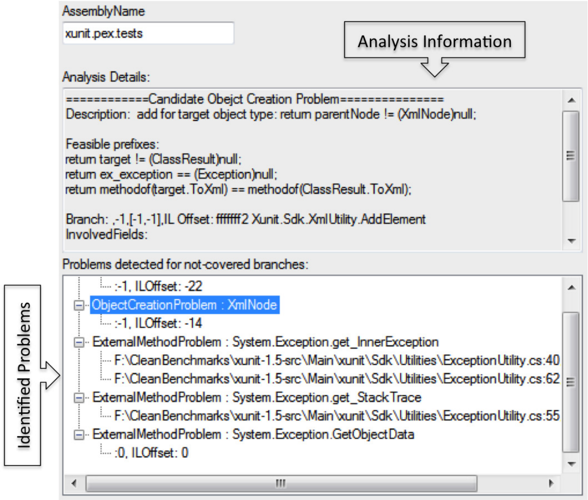
EMCP Candidate Identification. The problem-candidate identifier observes the method entry and exit events to identify EMCP candidates. If a method of the program under test is not instrumented by DSE, the method call is considered as an external-method call, being either a method call to system libraries or third-party pre-compiled libraries. Since the number of external-method calls can be large, the problem-candidate identifier considers as candidates only the external-method calls whose arguments have data dependency for program inputs.

OCP Candidate Identification. The problem-candidate identifier observes the program entry events to identify OCP candidates. OCP requires objects of a non-primitive type as program inputs, and the problem-candidate identifier ignores the program inputs whose type is a primitive type, such as `int`, `double`, and `boolean`. For program inputs of non-primitive types, the problem-candidate identifier marks the program inputs themselves and their fields as problem candidates of OCPs.

Forward Symbolic Execution. The forward symbolic executor leverages the DSE engine of Pex to perform forward symbolic execution by assigning symbolic values to elements of the identified problem candidates, including return values of external-method calls and program inputs of non-primitive types as well as their fields. We use the generated test inputs as program inputs during the forward symbolic execution.

Runtime Information Collection. The runtime information collected by the forward symbolic executor includes symbolic expressions of predicates in branch statements and uncaught exceptions. We collect symbolic expressions of predicates in branch statements for later computation of data dependencies of partially-covered branch statements. If some branch statements have data dependencies on prob-

[1] The release of Covana is available at http://research.csc.ncsu.edu/ase/projects/covana.
The demo video of Covana is available at http://research.csc.ncsu.edu/ase/projects/covana/covana.html



AssemblyName

xunit.pex.tests

Analysis Details:

```
============Candidate Obejct Creation Problem===============
Description:  add for target object type: return parentNode != (XmlNode)null;

Feasible prefixes:
return target != (ClassResult)null;
return ex_exception == (Exception)null;
return methodof(target.ToXml) == methodof(ClassResult.ToXml);

Branch: ,-1,[-1,-1],IL Offset: fffffff2 Xunit.Sdk.XmlUtility.AddElement
InvolvedFields:
```

Problems detected for not-covered branches:

```
    :-1, ILOffset: -22
ObjectCreationProblem : XmlNode
    :-1, ILOffset: -14
ExternalMethodProblem : System.Exception.get_InnerException
    F:\CleanBenchmarks\xunit-1.5-src\Main\xunit\Sdk\Utilities\ExceptionUtility.cs:40
    F:\CleanBenchmarks\xunit-1.5-src\Main\xunit\Sdk\Utilities\ExceptionUtility.cs:62
ExternalMethodProblem : System.Exception.get_StackTrace
    F:\CleanBenchmarks\xunit-1.5-src\Main\xunit\Sdk\Utilities\ExceptionUtility.cs:55
ExternalMethodProblem : System.Exception.GetObjectData
    :0, ILOffset: 0
```

Figure 2: The GUI component showing identified problems and detailed analysis information

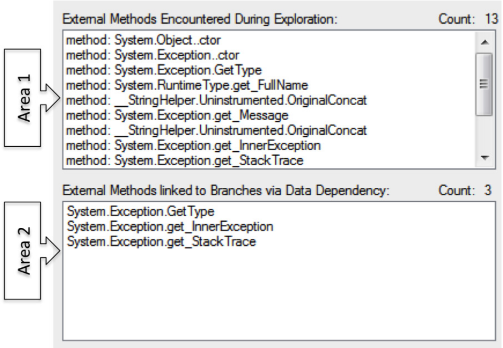
lem candidates, we can find constraints involving symbolic values in the collected symbolic expressions of predicates in branch statements. We collect uncaught exceptions during test execution, since uncaught exceptions thrown inside the execution of an external-method call can prevent Pex from exploring the remaining parts of the program after the call site of the external-method call.

2.2 Data-dependence Analyzer

The data-dependence analyzer consumes the coverage and runtime information collected by the forward symbolic executor, and computes data dependencies on problem candidates. Using the collected coverage, the data-dependence analyzer further prunes the problem candidates that none of partially-covered branch statements have data dependencies on. The GUI component presents the identified problems with the detailed analysis information.

EMCP Identification. The data-dependence analyzer identifies EMCPs using the computed data dependencies on EMCP candidates for their return values. If there exist data dependencies of partially-covered branch statement on an EMCP candidate, the data-dependence analyzer identifies such a candidate as an EMCP. To identify external-method executions that throw exceptions to abort test executions, the data-dependence analyzer further extracts the method calls from the collected stack traces of uncaught exceptions thrown during runtime. If such method calls contain any external-method calls, and the remaining parts of the program after the call site of the external-method call are not covered, the data-dependence analyzer identifies the extracted external-method call as an EMCPs that causes the remaining parts of the program not to be covered.

OCP Identification. The data-dependence analyzer identifies OCPs using the computed data dependencies on OCP candidates. If a partially-covered branch statement is data dependent for only program inputs, the data-dependence analyzer directly reports the program inputs as OCPs. However, if a partially-covered branch statement is data dependent for fields of program inputs, the data-dependence analyzer performs further analysis to identify which fields cause Pex not to achieve higher structural coverage. If a field cannot be assigned with an object directly by invoking a constructor or a public setter method of its declaring class,



External Methods Encountered During Exploration: Count: 13

```
method: System.Object..ctor
method: System.Exception..ctor
method: System.Exception.GetType
method: System.RuntimeType.get_FullName
method: __StringHelper.Uninstrumented.OriginalConcat
method: System.Exception.get_Message
method: __StringHelper.Uninstrumented.OriginalConcat
method: System.Exception.get_InnerException
method: System.Exception.get_StackTrace
```

External Methods linked to Branches via Data Dependency: Count: 3

```
System.Exception.GetType
System.Exception.get_InnerException
System.Exception.get_StackTrace
```

Figure 3: The GUI component showing the analysis of EMCP

the object state of the field can be changed only by invoking other public state-modifying methods of its declaring class. In this case, the data-dependence analyzer reports only its declaring class type as an OCP.

2.3 GUI Component

The GUI component of Covana takes as inputs the identified problems from the data-dependence analyzer and presents the problems with the detailed analysis information. Figures 2 and 3 show screenshots of the GUI component, which is built using windows forms [1]. In Figure 2, the GUI component displays the assembly name of the program under test and shows as a tree the identified problems with the related not-covered branches. When users select a problem, the GUI component presents the detailed analysis information of the selected problem. In Figure 3, Area 1 shows the encountered external-method calls and Area 2 shows the external-method calls that subsequent branches have data dependencies on.

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.

3. REFERENCES

[1] Windows Forms, 2002. http://windowsclient.net/.

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

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

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

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

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

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

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