Computing and Visualizing the Impact of Change with Java PathFinder Extensions

Eric Mercer
Brigham Young University
Computer Science Department
Provo, UT 84602-6576
egm@cs.byu.edu

Suzette Person
NASA Langley Research Center
1 S. Wright St. Mail Stop 130
Hampton, VA 23681-2199
suzette.person@nasa.gov

Neha Rungta
NASA Ames Research Center
Mail Stop 269-2
Moffet Field, CA 94035
neha.s.rungta@nasa.gov

ABSTRACT
Change impact analysis techniques estimate the potential effects of changes made to software. Directed Incremental Symbolic Execution (DiSE) is a Java PathFinder extension that computes the impact of changes on program execution behaviors. The results of DiSE are a set of impacted path conditions that can be efficiently processed by a subsequent client analysis. Path conditions, however, may not be intuitive for software developers without the context of the source code. In this paper we present a framework for visualizing the results of DiSE. The visualization includes annotated source code and control flow graphs indicating program statements that are changed and statements that may be impacted by the changes. A simulation mode enables users to also observe the impact of changes on symbolic execution of the program, by showing the changes to the path conditions as the user steps through the sequences of statements executed.

Categories and Subject Descriptors
H.5 [**Information Interfaces and Presentation**]: Miscellaneous

General Terms
Documentation

Keywords
Visualization, Program Analysis

1. INTRODUCTION
The impact of software changes can be widespread and unpredictable. Change impact analysis techniques [3] are used to estimate the potential effects of changes made to software. The results computed by these techniques are then used to guide automated client analyses, such as testing, verification, and debugging techniques, towards the parts of the program affected by the changes. Directed Incremental Symbolic Execution (DiSE) is a change impact analysis for characterizing the program behaviors impacted by changes to the code [6, 8]. DiSE estimates the impact of the changes on the source code using program slicing techniques (control– and data-dependence analyses), and then uses the impact sets to guide symbolic execution to generate path conditions that characterize impacted program behaviors. The impacted path conditions generated by DiSE consist of constraints on program inputs. Path conditions can be efficiently solved and analyzed by Satisfiability Modulo Theories (SMT) solvers [4] to facilitate subsequent analysis; however, path conditions can be difficult for software developers to process and understand.

In this paper, we present a framework for visualizing the results of a change impact analysis, and use the change impact results computed DiSE to demonstrate the features of the framework. The visualization includes annotated source code and control flow graphs indicating program statements that are changed, and statements impacted by the changes. A simulation mode enables users to visualize the impact of the changes on symbolic program execution, by stepping through the sequences of program statements executed to see when constraints are added to the path conditions.

The visualization tool presented in this work provides an integrated view of the results computed during the various steps performed by a change impact analysis. It also provides a way to map the program behaviors (path conditions) to source code locations. We believe it is easier for developers to make sense of results when the impacted program behaviors are mapped to the source code. The visualization is extremely valuable to us as developers of change impact analysis algorithms to understand the results produced by our technique. We also believe that this tool could be useful to software developers in general, by providing a better and more intuitive view of the impacts of program changes.

In the next section, we present a brief overview of the change impact analysis technique, DiSE, which is implemented as a Java PathFinder extension, *jpf-regression*. We discuss how information to facilitate the visualization is collected during the static analysis and symbolic execution phases of DiSE. In Section 3 we describe the inputs to the visualization framework and discuss the various visualization elements. Finally in Section 4 we discuss conclusions and future work.

2. DISE

Directed Incremental Symbolic Execution (DiSE) [6, 8] is a technique for characterizing the impact of software changes on program execution behaviors. The set of impacted path conditions computed by DiSE is useful for generating regression test inputs, and guiding other client analyses such as verification and debugging techniques. DiSE first estimates the impact of changes on the source code using static program slicing techniques. The impact sets generated during the slicing analysis are then used to guide symbolic execution to generate path conditions that characterize impacted program behaviors. In the remainder of this section, we briefly describe the DiSE technique. We then describe how we extended DiSE to collect the meta information used by the visualization framework. The reader is referred to [6, 8] for a more thorough explanation of the DiSE algorithm.

2.1 Inputs to DiSE

The inputs to DiSE are the source code for two versions of a procedure in programs P and P', and the results of a lightweight syntactic *Diff* analysis comparing the source code for P and P', e.g., textual or abstract syntax tree comparison. The results of the *Diff* analysis identify the change set – the set of locations in the source code that are different between the two versions. For program P', the *Diff* analysis marks source code lines that are *added*, *changed*, or *unchanged* with respect to P. Similarly, the analysis marks source code in P as *removed*, *changed*, or *unchanged* with respect to P'.

2.2 Static Impact Analysis

DiSE uses standard program slicing techniques (forward and backward), to generate the set of program locations that may be impacted by the actual changes. The slicing criteria are the program statements in the change set. Data- and control-flow analyses are used to identify impacted control statements – conditional branch statements that may be impacted by the change such that execution of these instructions leads to the generation of impacted program behaviors. The static analysis also identifies impacted write (assignment) statements – the values written at these locations may impact subsequent execution of conditional branch statements. Our implementation of DiSE supports both intra- and inter-procedural analyses to compute impacted statements.

2.3 Directed Symbolic Execution

To compute the impact of the changes on the execution behaviors of the modified version of the program, the impact set (of program locations) computed by the static impact analysis is used to direct symbolic execution of the modified version of the program and generate impacted path conditions. The impacted path conditions represent the program behaviors impacted by the changes. The algorithm is conservative and over-estimates the impacted behaviors. We have implemented a version of DiSE for analyzing Java programs in the *jpf-regression* extension to Java PathFinder (JPF). It uses the static impact analysis from the *jpf-guided-test* extension, and the Java PathFinder symbolic execution framework (SPF) [5, 7]. We use the Choco constraint solver [1] to check path feasibility during symbolic execution. During symbolic execution, a custom JPF listener is used to prune paths based on the reachability of impacted program locations from the current state. When there are no impacted (and unexplored) statements reachable on the current path, the listener instructs symbolic execution to backtrack, effectively pruning paths that do not execute at least one impacted program statement. At the end of symbolic execution, a symbolic summary containing the set of impacted path conditions for a given method is written to disk.

2.4 Extension to DiSE

In our original implementations of DiSE, we included an option to output the annotated control flow graphs built by the static analysis and showing changed and impacted nodes, in a format suitable for input to a graph visualization tool. These graphs were invaluable to us as developers of DiSE, enabling us to visualize the results of the static analysis; however, we only had the sets of path conditions generated by DiSE to represent the impacted program behaviors. Moreover, there was no way to link the results of the static analysis to the results of Directed Symbolic Execution.

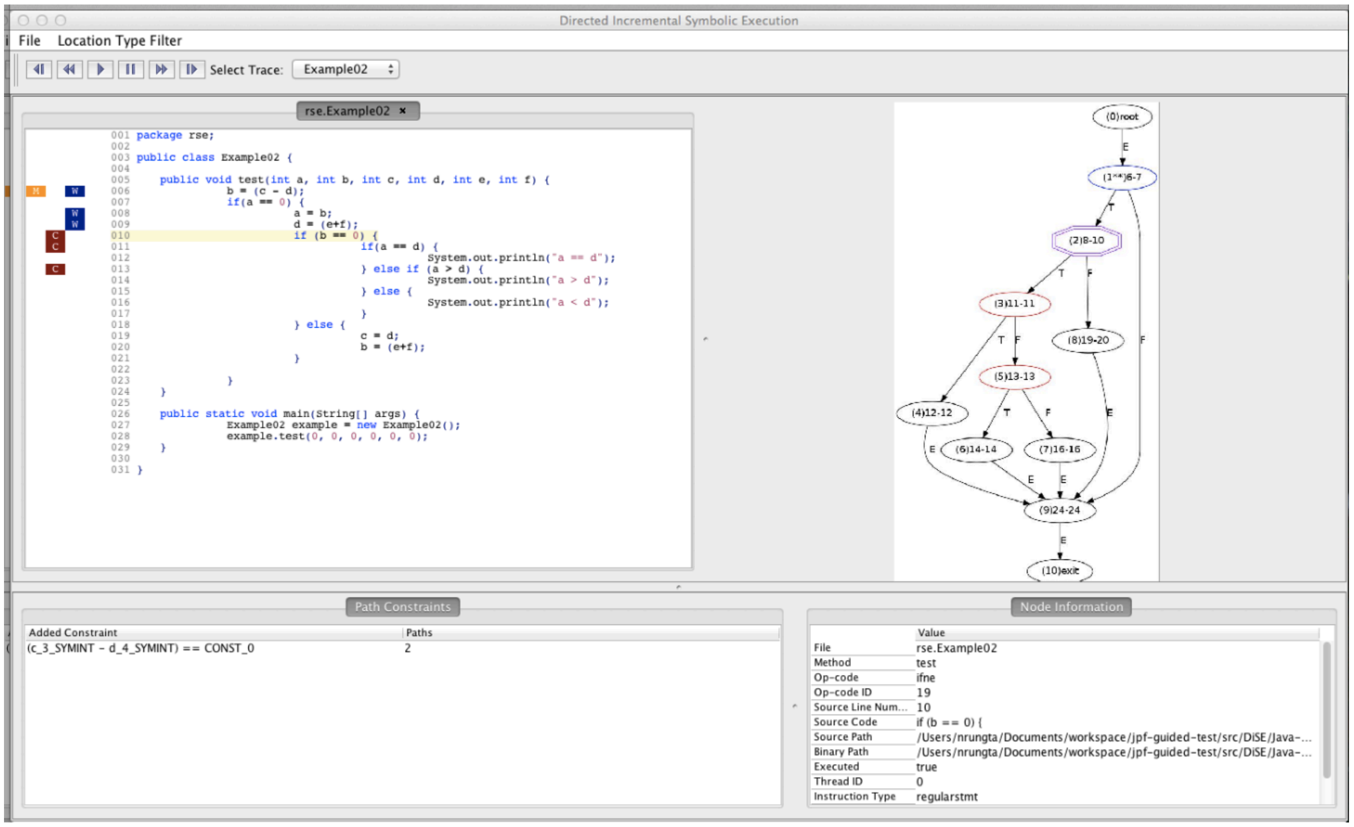
To create an integrated and more intuitive picture of the change impact analysis results computed by DiSE, we implemented a *DiSEVisualizationListener* to collect information during symbolic execution that can be used to visualize the impact of software changes on symbolic execution of the program. The *DiSEVisualizationListener* listens for instructions to be executed in methods identified as symbolic, and then records details about the instruction, e.g., the bytecode mnemonic, the source code, the source line number, and the execution order. When the instruction executed is an instance of an `if` instruction, the listener also records the constraint added to the path condition. At the end of symbolic execution, the listener outputs to an *XML file*, the information collected during symbolic execution. Each instruction executed in the symbolic method is also annotated with the results of the static impact analysis indicating if the instruction is changed, and whether the instruction is an impacted write statement, an impacted control statement, or a statement that is not impacted.

3. FRAMEWORK

The input to the visualization framework is an XML file containing the results computed by a change impact analysis and meta information about the analyzed software, e.g., control flow information relating program statements. The impacted and changed program statements are identified using specific color codes next to the source lines. The constraints generated during symbolic execution can be explored by using the simulation mode to step through the sequences of program statements executed during symbolic execution. Note that only change impact analysis techniques based on symbolic execution will have the option to run in the simulation mode. A screenshot of the DiSE visualization is shown in Fig. 1. In the remainder of this section we explain in detail, the input format to the visualization framework and each element of the visualization itself.

3.1 Inputs to the Visualization Tool

The change impact visualization framework is a post-mortem, offline analysis that can be used to visualize the results of



```
package rse;

public class Example02 {

    public void test(int a, int b, int c, int d, int e, int f) {
            b = (c - d);
            if(a == 0) {
                    a = b;
                    d = (e+f);
                    if (b == 0) {
                            if(a == d) {
                                    System.out.println("a == d");
                            } else if (a > d) {
                                    System.out.println("a > d");
                            } else {
                                    System.out.println("a < d");
                            }
                    } else {
                            c = d;
                            b = (e+f);
                    }

            }
    }

    public static void main(String[] args) {
            Example02 example = new Example02();
            example.test(0, 0, 0, 0, 0, 0);
    }
}
```

Added Constraint	Paths
(c_3_SYMINT - d_4_SYMINT) == CONST_0	2

	Value
File	rse.Example02
Method	test
Op-code	ifne
Op-code ID	19
Source Line Num...	10
Source Code	if (b == 0) {
Source Path	/Users/nrungta/Documents/workspace/jpf-guided-test/src/DiSE/Java-...
Binary Path	/Users/nrungta/Documents/workspace/jpf-guided-test/src/DiSE/Java-...
Executed	true
Thread ID	0
Instruction Type	regularstmt

Figure 1: Screenshot of the visualization of the results of DiSE on a simple example.

any change impact analysis. The visualization takes three inputs: the source code file, an XML file detailing the results of the change impact analysis, and an image of the control flow graph representation of the code.

The XML file is simply a list of nodes where each node maps to an instruction, e.g., bytecode, that was executed during symbolic execution. Each node has several attributes as explained below. The attributes for each `<node>` element are used to determine the color coding for the matrix, the constraints displayed in the path constraints pane, as well as the details presented in the node information pane. We use the following XML code fragment as an example:

```
<?xml version="1.0" encoding="utf-8" ?>
<graph id="ImpactAnalysis" index="0"
       tracetype="concrete">
<jpfstate id="none">
<node id="node_0"
      class="rse.Example01"
      ... />

<PCs>
<pc value="a_1_SYMINT == CONST_0" contained="1" />
</PCs>

</node>
...
</jpfstate>
<paths>
<path id="path_0" path="/Example01" />
<path id="path_1" path="/build/Example01" />
</paths>
</graph>
```

The XML file begins with a `<graph>` tag which contains a `<jpfstate>` element. The `<graph>` element has an ID, index, and tracetype that ranges over *concrete* and *abstract* (details to follow). The `<jpfstate>` element is simply a container for the list of nodes. Each `<node>` element in the XML file has the following attributes:

id: a string identifier used to give each bytecode a unique identifier.
class: the name of the class containing the method analyzed by the impact analysis. The tool currently supports display of intraprocedural analysis results only.
method the name of the method analyzed. This is also the expected name for the image file containing the control flow graph.
bytecode: the bytecode associated with the node.
bytecodeline: the bytecode line or index into the class file.
sourceline: the line number of the source code containing the bytecode.
tracetype: traces are either *abstract* or *concrete*. An abstract trace is not generated from an execution and typically does not contain path constraints. A concrete trace is generated from an execution and includes path

constraints.

sourcecode: the source code corresponding to the bytecode as it appears in the source file.

sourcecodefile: the name of the file containing the source code, i.e., the *.java* file.

bytecodefile: the name of the file containing the bytecode, i.e., the *.class* file.

executed: indicates if the bytecode is executed. The value is always false in an abstract trace. It can be either true or false in a concrete trace.

threadnumber: the thread ID.

type: an internal value that should always be *regularstmt*.

impacted: indicates how the bytecode is impacted by the modified line. The values ranges over *notImpacted*, *writeImpacted* (write), and *condImpacted* (conditional). A single bytecode can have only one impacted attribute.

modified: ranges over `true` and `false`. Indicates a changed source line.

srcpath: the fully specified path to the source code. The value ranges over a key defined in the `<path>` element in the XML as illustrated in the example above.

bytecodepath: the fully specified path to the class files. The value ranges over a key defined in the `<path>` element in the XML file as illustrated in the example above.

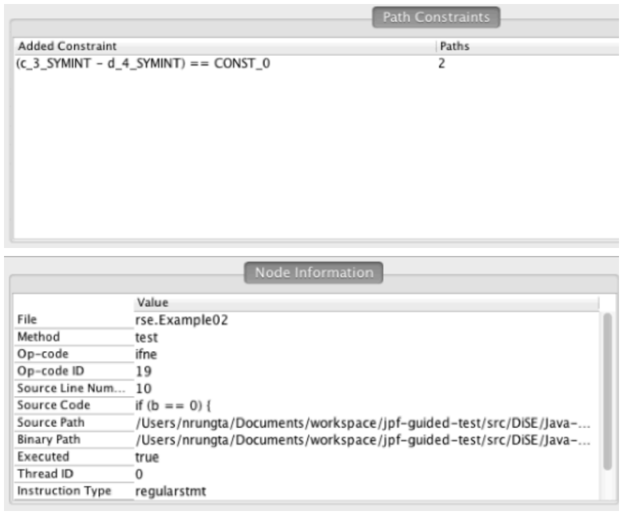
The sequence of the nodes in the XML file is assumed by the simulation mode to reflect the execution order of the program statements during symbolic execution. A node may optionally contain a list of path constraints as shown in the example above. The *contained* attribute is the number of total path conditions generated by the change impact analysis, e.g., DiSE, that contain the constraint identified by the *value* attribute. The XML file ends with the definition of the path keys referenced in the node attributes for the source and bytecode paths.

The last input to the visualization tool is the image file showing the control flow graph for the program. The tool assumes the image is in JPEG format, and looks for a file named *method*.jpg where *method* is the name indicated in the node definitions in the XML file. It should be the method analyzed by the change impact analysis. In the DiSE impact analysis, DiSE generates a *dotty* graph [2] of the control flow during static analysis. The *dotty* tool can be used to generate a JPEG image of the graph. The image is scaled to fit the visualization window, as shown in Fig. 1.

3.2 Adding visual cues to source code

Given a change set (of modified source lines), DiSE computes the set of impacted program instructions using static control and data dependence analyses. The impact set consists of impacted write statements and impacted conditional statements. Symbolic execution is directed toward the impacted statements. Program statements not impacted by the changes to the program *may* not be executed during symbolic execution.

The modified source lines, impacted write, impacted conditional, and non-executed source lines are identified using a color coded matrix. Each row in the matrix refers to a source code line. Each column indicates one of the above characteristics. If cell (i, j) is highlighted in the matrix, then source line i has attribute j. The rows i range over the natural numbers. The columns j range over {**M**,**C**, **W**, **N**} representing: [**M**]: modified, [**C**]: conditional impacted, [**W**]: write impacted, [**N**]: not executed.



Added Constraint	Paths
(c_3_SYMINT - d_4_SYMINT) == CONST_0	2

	Value
File	rse.Example02
Method	test
Op-code	ifne
Op-code ID	19
Source Line Num...	10
Source Code	if (b == 0) {
Source Path	/Users/nrungta/Documents/workspace/jpf-guided-test/src/DiSE/Java-...
Binary Path	/Users/nrungta/Documents/workspace/jpf-guided-test/src/DiSE/Java-...
Executed	true
Thread ID	0
Instruction Type	regularstmt

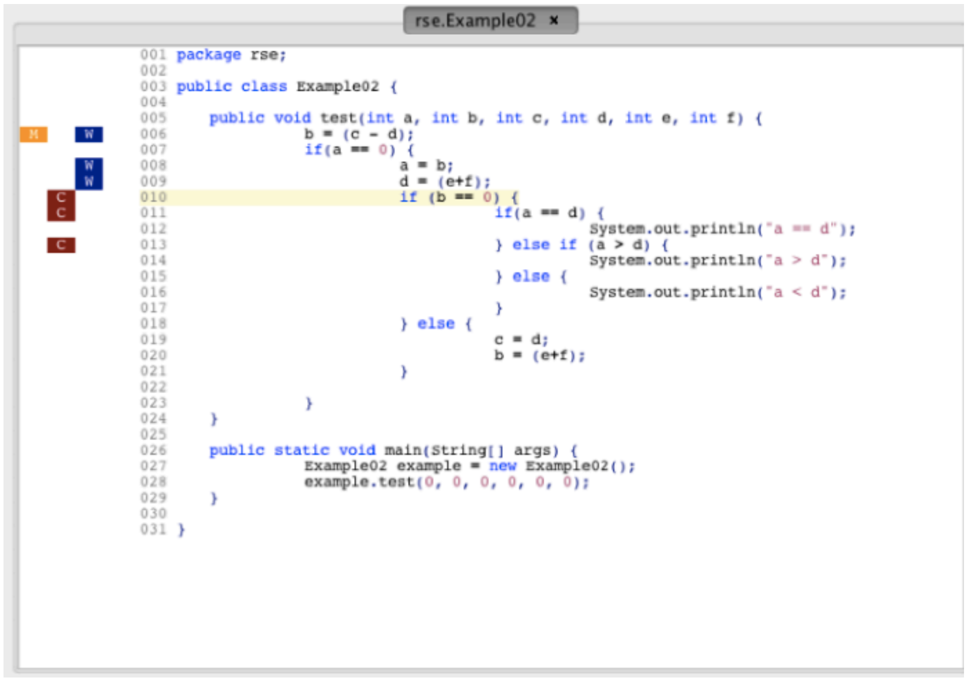
Figure 2: Bottom panes of the visualization that displays information during the simulation mode.

The color coded matrix is added as a visual cue next to the source code as shown in Fig. 3. For example, source line 006 has two color codes [**M**] and [**W**] indicating the statement is both modified and an impacted write statement. Lines 010, 011, and 013 have the [**C**] color code attached to them, indicating each is an impacted conditional statement. Since there are no program statements with the color [**N**] code, all the program statements in the example shown in Fig. 3 were executed during symbolic exeuction.

3.3 Simulation Mode of the Visualization

The search order of the analysis, i.e., the statement execution order during symbolic execution, added path constraints, and bytecode information are presented during the simulation mode using two extra display panes as shown in Fig. 2. The simulation mode highlights each line of the source code as it steps through individual byte-codes. Line 010 is highlighted in Fig. 3. The constraint that is generated at line 010 is shown in the left pane of Fig. 2 ($c\_3\_SYMINT - d\_3\_SYMINT == CONST\_0$). There are two paths in the execution that contain this particular constraint indicated by the column `Paths`. Identifying the contraints added at each conditional statement, we believe, can help developers understand the flow of impact information more easily, compared to just looking at the final set of all the path conditions.

The search order of the analysis is reflected in the step order of the source lines and individual bytecodes. For example, a depth-first analysis order shows a single path executed to the end of the method at which point it backtracks to the most recent impacted conditional to explore the unexplored branch. The simulation highlights the various source lines in such a manner until each path in the method is covered. The simulation is controlled similar to a debugger, using the



```
package rse;

public class Example02 {

    public void test(int a, int b, int c, int d, int e, int f) {
            b = (c - d);
            if(a == 0) {
                    a = b;
                    d = (e+f);
                    if (b == 0) {
                            if(a == d) {
                                    System.out.println("a == d");
                            } else if (a > d) {
                                    System.out.println("a > d");
                            } else {
                                    System.out.println("a < d");
                            }
                    } else {
                            c = d;
                            b = (e+f);
                    }

            }
    }

    public static void main(String[] args) {
            Example02 example = new Example02();
            example.test(0, 0, 0, 0, 0, 0);
    }
}
```

Figure 3: Source lines annotated with the color codes representing the modified and impacted statements.

controls at the top of the window.

As the simulation is played forward, added path constraints and the individual bytecode information for a given source line are displayed in two separate visualization panes. The path constraints show the terms added on each conditional statement. Included in the pane is an additional *contained* field which indicates the number of future paths that include the indicated constraint. Because each source line is composed of several bytecodes, the analysis simulation may take several steps, i.e., process multiple bytecodes before highlighting the next source line. The individual bytecodes processed are shown in the *Node Information* pane along with indications of whether the bytecode is modified, or an impacted write or conditional bytecode. As such, it is possible to learn which specific bytecodes caused the matrix encoding for a given source line.

4. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a tool for visualizing the results of DiSE, a change impact analysis we have implemented as a Java PathFinder extension. The development of the tool was driven by the capabilities of our analysis; however, the framework is capable of visualizing the results of other change impact analysis tools. The visualization framework includes a simulation mode that enables users to observe the impact of changes on symbolic execution by showing the changes to the path conditions as the user steps through the sequences of statements executed. The visualization also incorporates an image of the control flow graph generated by our DiSE implementations and an annotated view of the source code showing the results of the static impact analysis. Together, these visualizations provide software developers with an integrated view of the various results computed by DiSE and a more intuitive picture of how changes impact the program during symbolic execution.

The current version of the visualization framework supports the intra-procedural version of DiSE. In the future, we plan to extend the framework to support visualization of the inter-procedural version of DiSE. The current visualization framework has been used to present the results of small examples; extending the framework to support larger examples is also an important part of our future work.

5. REFERENCES

[1] Choco. `http://www.emn.fr/z-info/choco-solver/`. Accessed: 2012.

[2] Graphviz – Graph Visualization Software. `http://www.graphviz.org`. Accessed: 2012.

[3] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *ICSM '93*, pages 292–301, 1993.

[4] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[5] C. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE*, pages 179–180, 2010.

[6] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI*, pages 504–515, 2011.

[7] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, pages 15–25, 2008.

[8] N. Rungta, S. Person, and J. Branchaud. A change-impact analysis to characterize evolving program behaviors. In *ICSM, To Appear*, 2012.