© 2018 JIAYI CAO

AIDING AUTOMATED TESTING GENERATION PROCESS BY VISUALIZING DYNAMIC SYMBOLIC EXECUTION

BY

JIAYI CAO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Tao Xie

ABSTRACT

Significant progress has been made in the field of automatic test generation, and Dynamic Symbolic Execution (DSE) is among the most effective techniques in this field. Despite its success, DSE still suffers from various obstacles when applied on complex programs. Users of DSE-based testing tools often experience three main categories of problems: the object creation problem, external method call problem, and boundary problem. When these problems arise, the testing tools present little information about the problem causes, leaving the users in the dark. Furthermore, there is little visual (i.e., easy to digest) guidance readily available to solve the problems effectively. The lack of guidance is especially troublesome given that the tools do not scale well when the number of problems increases. As a result, the time needed to investigate the root causes of the problems is prohibitive.

To address such issue, in this thesis, we propose a visualization approach named PexViz. Our approach helps the tool users better understand and diagnose the encountered problems by reducing the large search space for problem root causes by aggregating information gathered through DSE exploration. In this thesis, we illustrate the benefits of the visualized information in assisting the tool users and our design decisions for such visualization. We provide a comparison between the proposed approach and two related previous approaches: IntelliTest, an industrial test generator available in Visual Studio 2015/2017, and an existing state-of-the-art visualization approach, SEViz. Our initial comparison results on example cases demonstrate the benefits of our proposed approach and its superiority over the related approaches.

To my family and my girlfriend for their love and support.

ACKNOWLEDGMENTS

I would like to especially thank my advisor, Professor Tao Xie, for his support and guidance. I would like to also acknowledge the help from Angello Astorga, Siwakorn Srisakaokul, Zhengkai Wu, and other current/former members from the Automated Software Engineering Group at the University of Illinois at Urbana-Champaign.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS vi

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND 4

CHAPTER 3 RELATED WORK 9

CHAPTER 4 MOTIVATING EXAMPLES 12

CHAPTER 5 PEXVIZ APPROACH 20

CHAPTER 6 IMPLEMENTATION 32

CHAPTER 7 CASE STUDY 34

CHAPTER 8 DISCUSSION 41

CHAPTER 9 CONCLUSION 44

REFERENCES 45

LIST OF ABBREVIATIONS

DSE	Dynamic Symbolic Execution
CFG	Control Flow Graph
IO	Input and Output
PUT	Parametrized Unit Test
API	Application Programming Interface

CHAPTER 1: INTRODUCTION

Manual testing is often time consuming and insufficient to achieve high code coverage. To address such issue, various techniques for automatic test generation and their supporting tools [1] are developed and adopted in industry. Among different techniques of automated test generation, Dynamic Symbolic Execution (DSE) [2–4] remains one of the most effective techniques. DSE starts with concrete execution of the program under test with a randomly generated input data or default input data (e.g., 0 for the integer input type). During the concrete execution, the executed path of the program is tracked, and a series of constraints are collected from the executed branching nodes (derived from branches taken upon conditionals) of the program. Such constraints are named *path condition*. Any input data satisfying the path condition will follow the same path as the executed path. Then DSE uses a search strategy [5] to select a branching node in the already-executed path to *flip*. Intuitively, flipping a branching node in the already-executed path refers to constructing a new path that shares the same prefix to the node with the already-executed path, but deviates and takes a different branch. A constraint solver is used to determine whether such resulting flipped path is feasible. If the constraint solver determines that the constraints in the new path condition for the flipped path are satisfiable and then further computes a satisfying value assignment for the program inputs, then we have found a new test input data to execute the program along the flipped path.

Although the DSE technique has been widely adopted as an effective way to automatically generate test input data, users of DSE-based tools often experience three main categories of problems [6–8] while applying the tools on complex programs. These three main categories of problems are the object creation problem, external method call problem, and boundary problem. (1) **Object creation problem**. When covering a branch in the program under test requires specific desired states of a non-primitive program input (e.g., a receiver object state or a non-primitive method argument), DSE may not be able to generate such desired object states resulted from a constructed sequence of method calls. (2) **External method call problem**. When covering a branch in the program under test requires specific desired return values of an external method call (e.g., a call to a method whose body is not explored by DSE due to its lacking access to the source or byte code, or due to its default configuration on exploring only method bodies within the project under test), DSE may not be able to trace from the constraints on the desired return values of an external method call to the required constraints on the method arguments for the external method call to produce the desired return values. Some common external method calls involved in such problem are

system API calls with environment dependencies such as network, database, file system API calls. (3) **Boundary problem**. When covering a branch in the program under test requires a large number of explored paths, DSE may not be able to cover such branch because its default exploration-resource allocation (such as the maximum number of explored paths allocated to path exploration) is not sufficient. Such problem often occurs for a program under test containing loops or complex string operations.

When these preceding problems arise, the testing tools present little information about the problem causes, leaving the users in the dark. Furthermore, there is little visual (i.e., easy to digest) guidance readily available to solve the problems effectively. The lack of guidance is especially troublesome given that the tools do not scale well when the number of problems increases. As a result, the time needed to investigate the root causes of the problems is prohibitive.

To address such issue, in this thesis, we propose a visualization approach named PexViz. Our approach helps the tool users better understand and diagnose the encountered problems by reducing the large search space for problem root causes by aggregating information gathered through DSE exploration. Our approach currently focuses on two main categories of encountered problems: the boundary problem and object creation problem. In particular, our approach provides visualization to summarize the path-exploration results by collapsing redundant exploration results through a Variant Control Flow Graph (VCFG) and then encoding information gathered from the DSE process on top of the VCFG. By iteratively interacting with the resulting graph, the users of a DSE-based tool can navigate through relevant information when diagnosing the encountered problems. In this thesis, we illustrate the benefits of the visualized information in assisting the tool users and our design for such visualization. To investigate the effectiveness of our approach, we provide a comparison between our approach and two related previous approaches: IntelliTest (derived from Pex [9, 10]), an industrial test generator available in Visual Studio 2015/2017, and an existing state-of-the-art visualization approach, SEViz [11]. Our initial comparison results on example cases demonstrate the benefits of our proposed approach and its superiority over the related approaches.

This thesis makes the following main contributions:

- A new approach of assisting DSE-problem diagnosis, named PexViz, including a visualization representation of a Variant Control Flow Graph (VCFG) and an algorithm for visualizing information gathered from the DSE process on top of such VCFG to help tool users diagnose encountered problems.

- An implementation of the PexViz approach in the form of a Visual Studio 2015 exten-

sion for IntelliTest, an industrial DSE-based tool.

- Initial comparison results on example cases for demonstrating the benefits of our proposed approach and its superiority over the related approaches.

The thesis is organized as follows. Chapter 2 introduces the background information for our proposed approach. Chapter 3 presents related work. Chapter 4 provides two motivating example cases for the approach. Chapter 5 illustrates the details of the approach. Chapter 6 presents the implementation details of the approach. Chapter 7 discusses initial comparison results on example cases for the approach and two related approaches. Chapter 8 discusses limitations of our current work along with future work to address these limitations. Finally Chapter 9 concludes the thesis.

CHAPTER 2: BACKGROUND

This chapter presents background information for our proposed approach: dynamic symbolic execution (DSE) (the test generation technique that our proposed approach is designed for), parameterized unit tests (the form of unit tests for which dynamic symbolic execution can be applied on), IntelliTest (the industrial DSE-based tool for which our proposed approach is implemented), and control flow graph (the basis of the visualization representation in our approach).

2.1 DYNAMIC SYMBOLIC EXECUTION

Symbolic execution [12–14] was first introduced as a formal, static analysis technique to assist test generation. In symbolic execution, the program under test is executed with symbolic inputs instead of concrete ones. For each execution, constraints are collected from each branch statement along the execution path. The conjunction of these constrains is a formula named *path condition*. Any input data satisfying the path condition will follow the same path as the executed one. Dynamic Symbolic Execution (DSE) [2–4] is a variation of symbolic execution. It is a mixture of both symbolic execution (which is traditionally static analysis) and concrete execution (which is dynamic analysis). As the normal concrete execution proceeds, symbolic execution also collects the constraints from conditions in branch statements that the execution process reaches, thereby constructing the path condition. DSE is often performed via multiple runs until the predefined halting condition is met such as reaching the maximum number of runs, the maximum number of executed branches, and the maximum timeout limit. The symbolic inputs are updated according to the constraint-solving solutions, to drive the executions to explore not-yet-covered portions of the program under test. Different paths are chosen to explore according to the search strategy [5] implemented in the DSE engine. The search strategy decides which branch to flip next (i.e., negating the constraint collected from the branch condition associated with the branch), therefore forcing the program to enter both the "true" and "false" branches of the branch condition at different runs. The number of times that a branch is flipped is denoted as **flip count**. Manually specified assumptions are specifications of conditions for the DSE engine to generate only inputs satisfying such conditions, e.g., a parameter should not be null or empty, or should contain a specific character.

As discussed in Chapter 1, three major problems that prevent DSE-based tools from achieving high code coverage are (1) **object creation problem**, (2) **external method**

call problem, and (3) **boundary problem**. Although the tools cannot effectively identify these problems and resolve them, tool users can provide guidance to the tools to improve code coverage. For example, to solve object creation problems, the tool users can create factory methods [9,15], including method sequences [16–18] to construct complex objects to cover those branches whose coverage requires specific desired states of the objects. External method call problems can be solved by instrumenting external method calls or provide mock objects [19–24] to simulate the method behavior. Limiting loop iterations [25] or increase the resource-limit boundary can help with solving boundary problems.

Listing 2.1: An example parameterized unit test in C#

```
using System;
using Microsoft.Pex.Framework;
using Microsoft.VisualStudio.TestTools.UnitTesting;

[PexClass(typeof(Program))]
public partial class TestClass {
  [PexMethod]
  public int WhileLoopTest([PexAssumeUnderTest]Program target,
   int[] a, int b)
  {
     int result = target.WhileLoop(a, b);
     return result;
  }
}
```

2.2 PARAMETRIZED UNIT TEST

Parametrized unit tests [26–28] (PUTs) are a general form of unit tests, as an extension and generalization to conventional closed unit tests, which take no parameters. PUTs take parameters and test the designed logic with values generated for the parameters. With PUTs, developers have control over the parameters: values for the parameters can be generated manually or automatically through a DSE-based tool. Conventional closed unit tests can be (re-)obtained by instantiating PUTs with values generated for the parameters. In our proposed approach, PUTs are the main program that the users of our approach need to write and modify on. In particular, PUTs are manually written and marked with the property attribute of `PexMethod` to be recognized by a DSE-based tool such as Pex [9, 10]

and its commercial version named IntelliTest. A PUT is executed symbolically by assigning symbolic variables to parameters. Each run of DSE results in a path condition, which can be solved to find a solution that instantiates the parameters of the PUT with concrete values. Listing 2.1 shows an example PUT written in C# for IntelliTest to generate test inputs for the WhileLoop method.

2.3 INTELLITEST

Our approach is developed for and integrated with IntelliTest, an automated test generation tool for .NET programs, shipped with the Microsoft Visual Studio 2015/2017 Enterprise Edition. IntelliTest is the commercial version of Pex [9, 10], a DSE-based test generation tool. A standard usage of IntelliTest is to configure a test project in Visual Studio to include a reference to Microsoft.Pex, write a PUT (in the test project) with the attribute property of `PexMethod` similar to the example in Listing 2.1, and run IntelliTest's exploration process on the PUT. For PUTs with parameters of complex objects, IntelliTest automatically guesses the structure and creates factory methods to generate the objects. However, the aforementioned problems in DSE still apply in the IntelliTest context, and our approach aims to provide assistance to problem diagnosis when using IntelliTest.

```
11      public class Program
12      {
13          public static void Main(string[] args)
14          {
15              Console.WriteLine(Foo(-10));
16          }
17          public static void Alert(string x){
18              Console.WriteLine(x);
19          }
20          public static string Foo(int x)
21          {
22              if ( x == 0 ){
23                  return "None";
24              }
25              else if ( x < 0 ){
26                  if ( x % 2  == 0 ){
27                      return "Even Neg";
28                  }else{
29                      return "Odd Neg";
30                  }
31              }else{
32                  for(int i = 0; i < x ; i++){
33                      Alert("Plus one");
34                  }
35                  return "Positive";
36              }
37          }
38      }
```

Figure 2.1: Example `Foo` method

2.4 CONTROL FLOW GRAPH

A control flow graph [29] (CFG) for a program refers to a graph where all paths on the graph represent the possible execution ordering of the statements within the program. Each node in the CFG represents one statement or one block of statements in the program, and each edge represents the jumps between different statements or statement blocks. Typically the CFG also contains entry nodes responsible for managing the entering into the CFG, and exit nodes responsible for managing the ending. Figure 2.2 shows an example CFG for the `Foo` method in Figure 2.1. A CFG is the basis of the visualization representation in our approach.

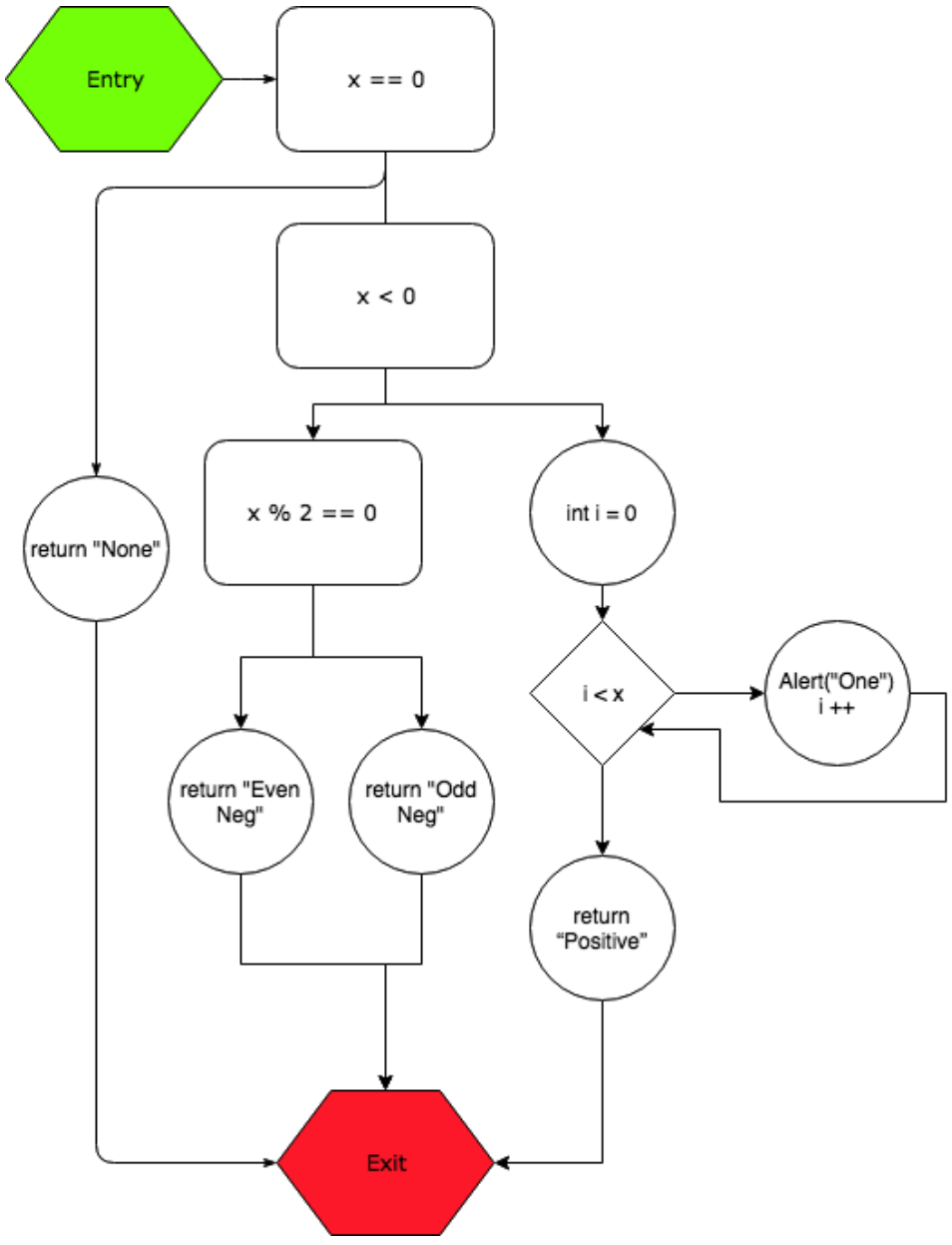

Figure 2.2: Example control flow graph for the `Foo` method in Figure 2.1

CHAPTER 3: RELATED WORK

Identifying external method call problem and object creation problem for DSE. Xiao et al. [6, 7] identify and summarize main problems faced when using DSE to generate test inputs, and develop a tool named Covana to help address such problems. Through their study, they find two major causes of insufficient statement or branch coverage: the external method call problem (EMCP) and the object creation problem (OCP). Although their tool can identify such problems, their tool's results are shown to users in textual outputs. The output format provides limited guidance to the users in addressing the problems because the output format is not easily understandable and contains long content. In addition, after initially attempting to address the problems, the users still need to spend time reading the textual outputs to verify that the problems are indeed solved. In contrast, using a visualization approach such as the one in this thesis can help the users quickly identify potential root causes and reduce the investigation time from the iterative process of reading long textual outputs.

Visualizing DSE information. SEViz [11] is a visualization approach to help tool users identify problems during DSE. It "interactively visualizes symbolic executions in a form of symbolic execution trees". The input of the approach is the program under test and PUTs manually written for IntelliTest to generate test inputs. The output of the approach is a directed acyclic interactive graph that combines multiple runs of the symbolic execution trees into a single tree. The general information gathered through IntelliTest's DSE process is organized into interactive windows that the users can easily access through clicking on the graph. The color schema of the nodes also enables the users to quickly identify the execution path that fails to generate new test inputs. However, there are two main limitations of the approach. First, the approach is not scalable to the program under test that contains loops and recursions due to path explosion. Second, the approach does not provide sufficient information on identifying problem causes, but relies on the users' time-consuming investigation to explore possible problem causes. In contrast, our approach is scalable to a large program under test by reducing the infinite size of the execution tree to the size of a VCFG for the program. Our approach also consolidates information such as flip counts to help the users identify potential causes. The prototype [30] of a Visual Studio plug-in for SEViz is well implemented and used as a baseline for evaluating our PexViz approach.

Viscovery [31] is a visualization approach to visualize the problem-analysis results and the coverage information for a DSE tool such as Pex [9, 10]. The input of the approach

is the program under test, and the output is multiple views containing different diagnosis information on the DSE exploration result. The implementation also integrates the Covana tool [6, 7] into its Problem View for listing out problems that Covana discovers during DSE exploration. The contextual view and code view utilize visualization techniques to help the users quickly identify abnormal flip counts and code coverage areas. The idea of using a color schema to represent the percentage of covered and flipped code areas is closely related to the ideas in our approach. A main limitation of their approach is that the color schema is directly applied to the source code, so the users still need to navigate through complex code structure to identify points of interest to investigate. In contrast, similar to Jinsight's execution view [32], which lets the users view execution sequences of multiple method calls in a single graph, our approach visualizes the DSE information in a VCFG such that DSE information across different files is consolidated in the VCFG, without requiring the users to browse through multiple files and therefore improving the users' efficiency of problem diagnosis. Furthermore, although Viscovery is developed for .NET programs being tested by Pex, its prototype is implemented as an Eclipse plug-in and is loosely integrated with Pex in Visual Studio. Such loose integration causes usability issues and inconvenience for the users. In contrast, our approach is developed as a Visual Studio extension tightly integrated with IntelliTest.

Visual interactive debugging based on symbolic execution. Hahnle et al. [33] develop a visualization approach to use symbolic execution trees as a debugging application to help debugging in software development instead of aiding test generation. Their idea presents a concise representation to visualize loops and recursions with the given contract specifications and loop invariants. The approach is highly robust to implementation changes because it separates the contract of a method from its implementation. This important feature enables developers to write acceptance tests for externally developed libraries without knowing implementation details. The final output of the approach is a tree whose nodes are composed of different types of information such as termination condition, loop invariant, branch name. The information includes different precondition setups but is represented in a relatively chaotic manner. Using such graph compression techniques [34] does not directly help the tool users and may cause important information to be ignored by the users. Although the tool is not deliberately developed to help generate test inputs, it gives ideas on what a good visualization of symbolic execution should be. More detailed and organized information gathered from DSE is needed to make their approach more succinct and lossless in terms of information that helps the users to make decisions. In contrast, our approach provides multiple sources of useful information such as flip counts and halting reasons of the DSE process in addition to the VCFG to help the users make decisions.

Visualizing test information to assist fault localization. Jones et al. [35] develop an approach that uses colors to visually show the number of times of a statement (within a method) that appears in a failing or passing test. Using color-coded graphs can enrich the content of the visualization and make the problem-identification stage shorter. Given a project that contains an extensive number of tests, the large number of times that tests fail at a certain statement can indicate that statements near that statement have potential faults for causing the tests to fail. By taking advantage of the HSV color space [36], which is more intuitive to human, the color component represents the percentage of passing tests and the brightness component represents how high percentage the type of dominating test results (pass or fail) is. The visual cues are helpful to the users because as the problem scales, human effort to find problem causes from textual data is increased substantially due to too much redundant information in the textual data. However, their approach is not to help address problems in DSE for test generation, which is focused by our approach.

CHAPTER 4: MOTIVATING EXAMPLES

This section presents two motivating examples of problem diagnosis with existing approaches for a DSE-based test generation tool. The first example is on problem diagnosis by using IntelliTest shipped with Visual Studio as it is. The second example is on problem diagnosis by using IntelliTest with the SEViz [11] visualization extension.

The example method under test `Resolve` used in this chapter is from from the GitSharp [37] project, an open source C# implementation of Git for .NET framework and Mono. GitSharp is a fairly complex mid-size library for Windows desktop applications using the .NET framework. As the example method under test, we choose the `Resolve` method within the class `Repository` under the `GitSharp.Core` package because this method is one of the most heavily investigated examples for SEViz [11]. This method takes one parameter `revision` and outputs the identifier in type `ObjectId` of the `Git` object in the current Git repository; such identifier corresponds to the revision number. The source file of the method that we use as the example method under test is from revision e5dfc76 from the GitHub GitSharp project repository. This method contains 289 lines of code composed of 29 `if` statements, 8 `while` loops, 7 `for` loops, and 2 `switch` statements with 14 different cases.

Listing 4.1: The PUT created for the `Resolve` method

```
[PexMethod]
public ObjectId ResolveTest(
 [PexAssumeUnderTest]Repository repository, string revision)
{
        ObjectId result = repository.Resolve(revision);
        return result;
}
```

4.1 USING INTELLITEST ONLY

We create a PUT as shown in Listing 4.1 for the `Resolve` method, and run IntelliTest with its default configuration. Given the PUT, the DSE process takes two runs, with the console warning in IntelliTest as shown in Figure 4.1. We next illustrate the detailed steps of conducting problem diagnosis when running IntelliTest against the PUT.

1. Creating a basic PUT and running IntelliTest with its default configuration result in generating no test inputs and achieving 0% block coverage. The exploration halts right

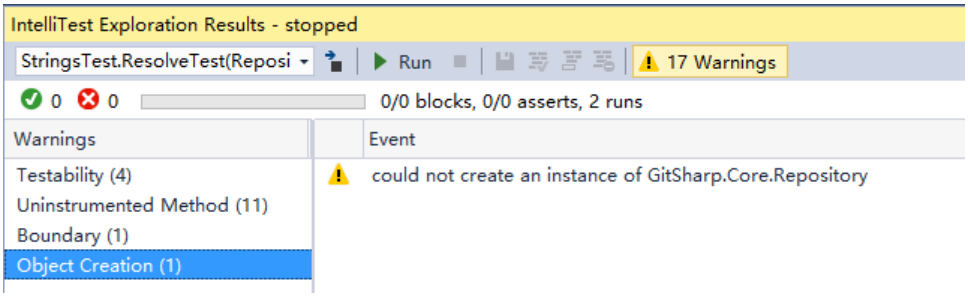

Figure 4.1: Example console warning in IntelliTest

away because IntelliTest is not able to create a `Repository` object and fails to guess how to create a `Repository` object.

2. After reading the error and warning messages produced by IntelliTest, we construct a factory method as in Listing 4.3 for the class `Repository`, but IntelliTest still shows no improvement. After reading online resources/documentation and consulting experts on how to improve the code coverage under such situation, we are able to adjust the settings in the PexAssemblyInfo.cs file and add settings related to assembly-level instrumentation as in Listing 4.2, and then IntelliTest is able to generate 14 test cases and achieve 18% block coverage.

Listing 4.2: PexAssembly.cs

```
// Microsoft.Pex.Framework.Settings
[assembly: PexAssemblySettings(TestFramework =
  "VisualStudioUnitTest")]

// Microsoft.Pex.Framework.Instrumentation
[assembly: PexAssemblyUnderTest("GitSharp.Core")]
[assembly: PexInstrumentAssembly("ICSharpCode.SharpZipLib")]
[assembly: PexInstrumentAssembly("Tamir.SharpSSH")]
[assembly: PexInstrumentAssembly("System.Core",
  InstrumentationLevel = PexInstrumentationLevel.Excluded)]
[assembly: PexInstrumentAssembly("Microsoft.VisualBasic",
  InstrumentationLevel = PexInstrumentationLevel.Excluded)]


// Microsoft.Pex.Framework.Creatable
[assembly: PexCreatableFactoryForDelegates]
```

```
// Microsoft.Pex.Framework.Validation
[assembly:
  PexAllowedContractRequiresFailureAtTypeUnderTestSurface]
[assembly: PexAllowedXmlDocumentedException]
```

Listing 4.3: Factory method created for the `Resolve` method

```
public static partial class RepositoryFactory
{
        private static DirectoryInfo trash = new DirectoryInfo
                (Path.Combine("target", "trash"));

    [PexFactoryMethod(typeof(Repository))]
    public static Repository Create()
    {
        String uniqueId = Guid.NewGuid().ToString();
        String gitdirName = "test" + uniqueId + ("/")
                        + Constants.DOT_GIT;
        DirectoryInfo gitdir = new DirectoryInfo(
                Path.Combine(trash.FullName, gitdirName)
        );
        Core.Repository db = new Core.Repository(gitdir);
        return db;
    }
}
```

3. Given that the so-far-achieved block coverage ratio is still not desirable, after carefully examining the source code and report generated by IntelliTest, we notice that there are repeated warnings about file-related library/method usages. Among 17 different warnings, the most frequent class names are listed as below:

   (a) `Directory`

   (b) `FileSystemInfo`

   (c) `Path`

   (d) `PathHelper`

We speculate that DSE encounters the external method call problem since all of the preceding classes and their methods interact with the file system. It is a reasonable speculation because the nature of the GitSharp project is to provide a .NET interface to manage project files on disk under the Git version control. We further create a number of mock objects for the preceding classes and involved methods to isolate the dependency of the file system. It takes about an hour for the author to write the correct mock objects and integrate them into the test project. After running IntelliTest again on the PUT, we still observe the block coverage to be as low as 23%.

4. After carefully examining the current generated test inputs, we find that only few of them contain valid strings (i.e., ones that can be regarded as a revision id) as the method argument. To figure out where the unflipped branches are located and why the branches are not flipped, we go through every line of the source code and find that some of the unflipped branches are expecting the revision id to contain special characters such as "@" or "^" or special substring sequence such as "tree" or "commit". Then we add assumptions to guide constructing strings that may contain some of the special characters such as `PexAssume.IsTrue(revision.StartsWith(''@''));`, and we finally achieve 56% block coverage after running IntelliTest again.

4.2 USING INTELLITEST WITH SEVIZ

In this section, we use the same PUT shown in Listing 4.1 but run IntelliTest with the SEViz plugin on the PUT. It is expected that SEViz should be helpful in diagnosing the causes for low code coverage.

1. Running IntelliTest with the default configuration on the PUT gives 0% block coverage. The exploration has only 1 run and the output console of IntelliTest shows 5 warnings. Among the 5 warnings, there are 2 warnings of the object creation problem, 2 warnings of the uninstrumented method problem (i.e., the external method call problem), and 1 runtime warning. In the visualization produced by SEViz, there are two red nodes without any edges on the graph. Upon clicking on the nodes, we are able to observe the status of the nodes as "AbandonedBecauseOfAssumptionViolation". The provided information does not help us make any decisions.

2. According to the knowledge that we learn (as described in the preceding section), we manually create some factory methods and IntelliTest still achieves 0% block coverage. In the SEViz graph, we observe two red nodes and one green node. Upon clicking

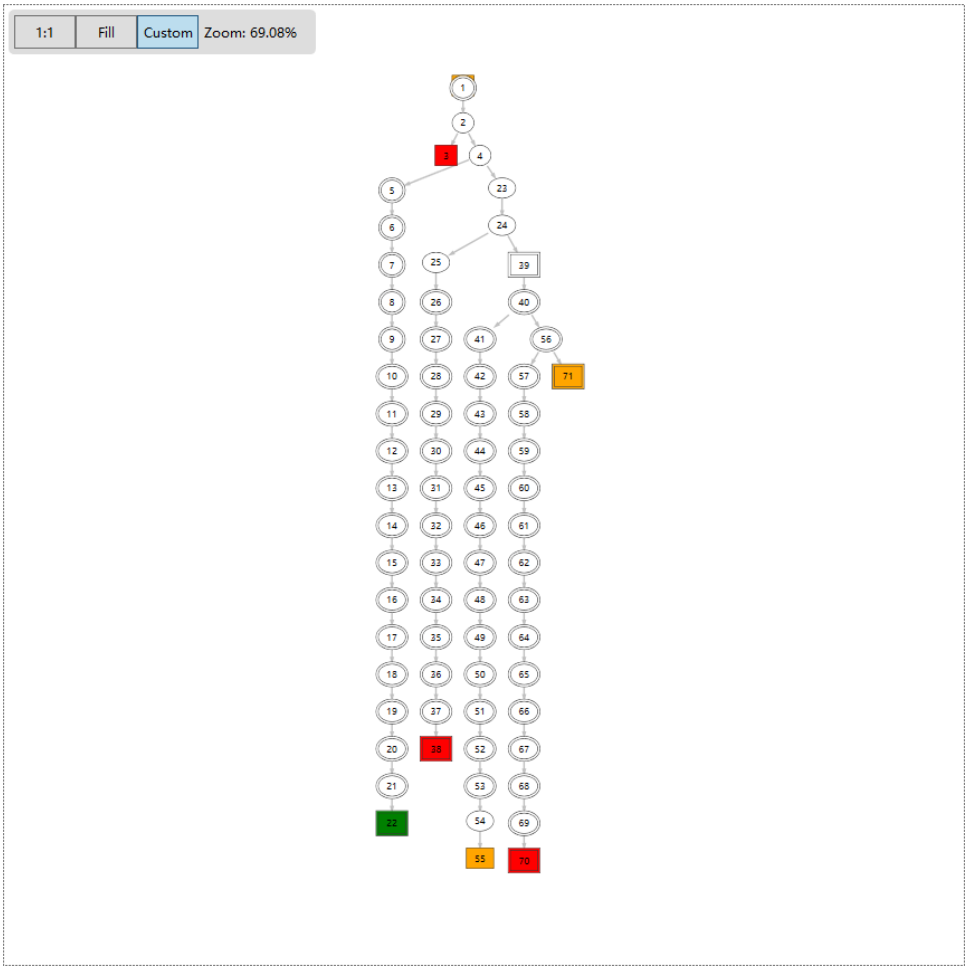

Figure 4.2: SEViz visualization when IntelliTest achieves 18% block coverage

on the green node, we observe that a new test case is generated for triggering a null-pointer exception. The visualization is still not helpful yet. However, once we add the instrumentation settings as in Listing 4.2, IntelliTest achieves 18% block coverage. Then the SEViz visualization shown in Figure 4.2 becomes more interesting.

3. According to the visualization graph shown in Figure 4.2 produced by SEViz when IntelliTest achieves 18% block coverage, we observe that the four long consecutive paths composed of white nodes seem to be taking a large amount of computing resource. Upon clicking on different white nodes, we are able to examine the corresponding source code. After clicking through 30 nodes and examining the corresponding source code, we are able to identify that the majority of the execution nodes are about flipping a branch related to a file-system call. The branch is checking whether a certain file exists on disk and keeps failing on coming up with the name of a file that exists on disk. After we isolate file-system calls with mock objects, IntelliTest achieves 23% block coverage.

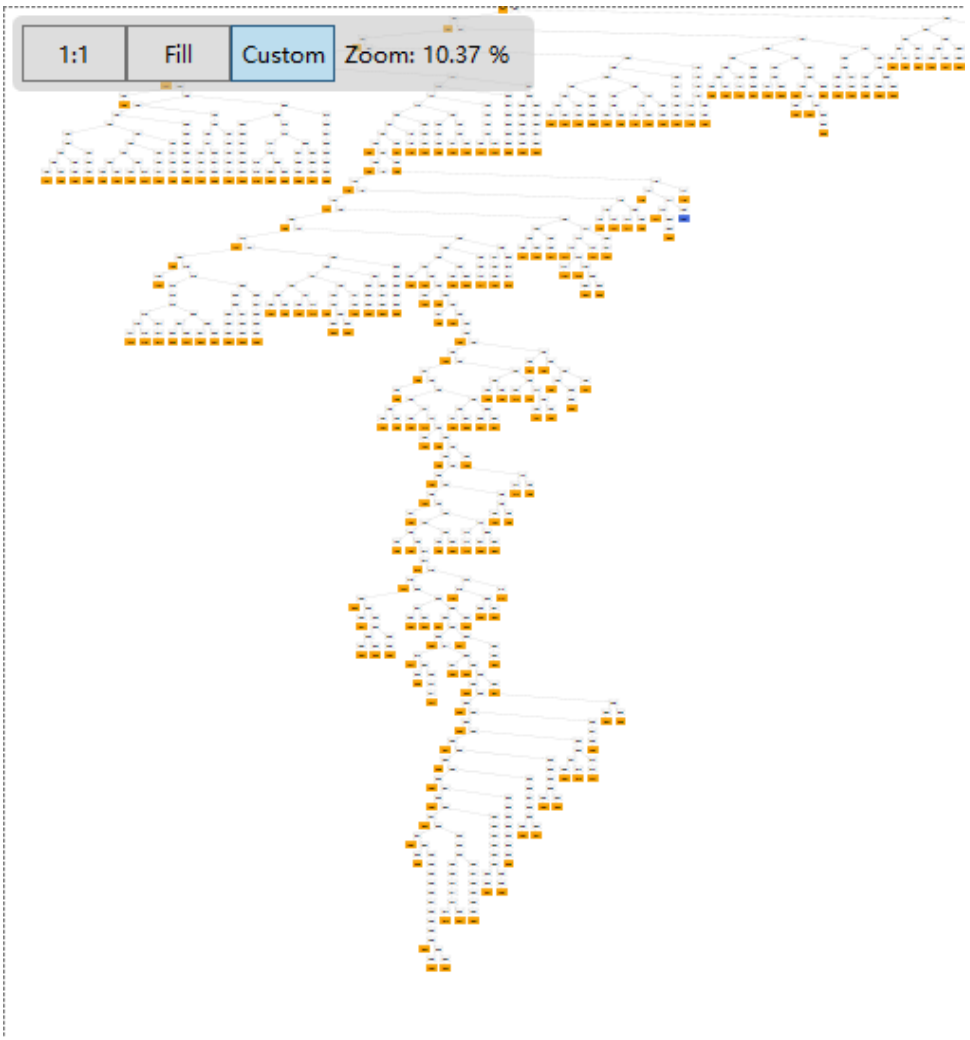

Figure 4.3: SEViz visualization when IntelliTest achieves 23% block coverage

4. In the new visualization graph (shown in Figure 4.3) produced by SEViz when IntelliTest achieves 23% block coverage, there are too many nodes and it is really hard to inspect an individual node without zooming in. By zooming in the graph, we find that it is difficult to keep track which execution path that we are currently investigating. We have to constantly go back and forth in different areas of the graph. Clicking on a node will transfer the window and mouse focus to the source code window, making the investigation process chaotic with limited screen space, but we manage to identify that most of the nodes are resulted from trying to flip branches that are looking for specific patterns in the revision id. Thus, we are able to identify the needed string pattern from the source code and write an assumption to enforce the needed constraint on the method argument. At this point, IntelliTest reaches the same coverage as using IntelliTest alone (56% block coverage).

4.3 DISCUSSION

Designing a useful visualization system is challenging [38]. The system designer must understand what information can be useful to the users. After using the two preceding existing approaches to try to increase the block coverage on the `Resolve` method in the GitSharp project, we next discuss the difficulties faced in the investigation process and what kind of information could help shorten the needed time and focus on the right locations to diagnosis the faced problem.

(a) The visualization should be scalable on programs with a large execution tree.

A SEViz visualization graph can often have a huge number of nodes. A huge graph can be produced by SEViz for even a simple program with one or two loops because loops can be executed for a large number of times. The size of the graph should not be proportional to the number of all the execution nodes explored by the DSE engine. Also, majority of the information is repeated in the case of loops, and the execution keeps executing the same branch. In such situation, visualizing all the execution nodes causes the graph-size explosion, yet bringing little new information for problem diagnosis.

(b) It is desirable for the visualization to provide visual correspondence between a graph node and its corresponding branch condition.

In SEViz, clicking on a node will jump to the corresponding source code. While such navigation is helpful, it is still difficult to check back and forth in different windows. There is no direct information on the nodes themselves to indicate which node corresponds to which line of source code, and it is difficult to remember such correspondence during the investigation process.

(c) It is desirable for the visualization to provide more information on which branches are flipped, which ones are not flipped.

In both IntelliTest alone and IntelliTest with SEViz, it is difficult to know which branches in the source code have been flipped (and how many times) and which ones have not been. Such information is critical for the users to conduct problem diagnosis for low code coverage. By quickly identifying an unflipped branch, the users can quickly inspect its neighboring nodes to find out potential causes.

(d) It is desirable for the visualization to represent the relationship between nodes.

In SEViz, most nodes are connected with edges, but there is no formal definition of the edges, and these edges do not carry much useful information. Some example

questions arise. How does one subtree relate to another subtree? Do the subtrees all start from the same parent branch? How is a node different from another one? The answers to these questions should be easily available to the users, and more information on the edges can be conveyed in the visualization.

To satisfy the preceding requirements and address the limitations of the existing approaches, we design a new visualization approach to present information critical for problem diagnosis in a scalable and comprehensible way. In the next chapter, we present our approach.

CHAPTER 5: PEXVIZ APPROACH

The overview of our approach is shown in Figure 5.1. Our approach consists of three components: the Variant Control Flow Graph (VCFG) generator, the exploration-data augmentor, and the graph visualizer. The CFG generator reads the program source code and transforms it into a VCFG representation. In a VCFG, a typical node corresponds to a branch statement in the program source code, and a directed edge between the starting node and the ending node indicates the control flow from the branch statement represented by the starring node to the branch statement represented by the ending node. The exploration-data augmentor is invoked by the IntelliTest exploration runtime and gathers useful information to augment the VCFG. Finally, the graph visualizer reads the output VCFG and generates an interactive visualization front-end to present information to the users.

In the rest of the thesis, we use the following following terminology. **DSE exploration** refers to the whole DSE process to search for possible inputs to achieve higher code coverage. An **execution path** refers to a path explored during a single run of the DSE exploration. The **path condition** for an execution path consists of predicates gathered from the executed branches; executing any inputs that satisfy the path condition results in the same execution path. A **branching node**, being a dynamic concept, is the execution node derived from an executed branch in an execution path. A **branch statement**, a static concept, is the conditional statement in source code. The **branch condition**, a static concept, for a branch statement is the boolean predicate from the conditional in the branch statement. A **branch**, a dynamic concept, is an outgoing side of executing a branch statement, depending on the corresponding branch condition's evaluation result. A **predicate** is a Boolean expression whose value can be true or false depending on the values of the variables in the expression.

5.1 VCFG GENERATOR

The VCFG generator reads and parses the program source code and generates a VCFG. The main functionality of the VCFG generator is recursively scanning the specified program source code, starting from the method under test specified in the PUT and collecting the method's branch statements and the possible execution paths between them.

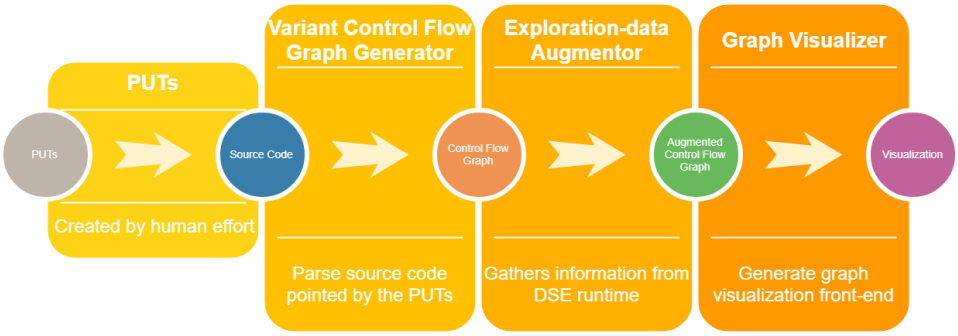

Figure 5.1: Components of PexViz

5.1.1 Generating VCFG Nodes

Depth-first search is implemented in the component to gather branch statements and transform them into VCFG nodes. Each level of the search space is a block of statements. The entry point of the method under test is represented as the entry VCFG node of the graph. The VCFG nodes representing the enclosed branch statements of the enclosing branch statement are regarded as a subgraph connected to the VCFG node corresponding to the enclosing branch statement. In this process, the component omits any non-branch-type statements. Basically, the graph includes only statements that are among `if`, `while`, `for`, and `switch` statements. After each branch statement is recorded in the graph as a VCFG node, the branch statement's corresponding metadata information, such as the line number and branch condition, is extracted and recorded in the VCFG node.

5.1.2 Generating VCFG Edges

We create a VCFG edge labeled with "True" between the starting VCFG node representing a branch statement b_s to the ending VCFG node representing the first branch statement b_e within b_s's true-branch block. If b_s contains a false-branch block, a VCFG edge labeled with "False" is created similarly.

To capture statements' sequential execution sequence, we use VCFG edges without labels. For every level of the recursive searching process, the component records a series of VCFG nodes as leaf positions because their execution paths continue down to their enclosing branch statement's next statement when they finish executing. We create VCFG edges between such VCFG nodes at leaf positions and the first VCFG nodes after their enclosing branch statement in the source code. For example, in Listing 5.1, after we finish searching with the branch statement at Line 1, we mark VCFG nodes corresponding to Lines 2 and 6 as leaf VCFG nodes. When we start to search at Line 10, we create two VCFG edges with no labels

between the leaf VCFG nodes and the VCFG node corresponding to Line 10. For loop types of branch statements, we create VCFG edges labeled with "loop" between the nodes at leaf positions and the overarching branch VCFG node.

Listing 5.1: Code snippet with simple branch statements

```
1  if( x== 1){
2      If( y==1 ){
3          a++;
4      }
5  }else{
6      if( z==1 ){
7          b++;
8      }
9  }
10 If( v==1 ) c++;
```

5.1.3 Dealing with multiple methods

The classic CFG involves a single method, but constructing a fully connected CFG that shows all possible paths between all invoked methods in branch statements is necessary for it to be compatible with DSE results. A method may have multiple return statements, or the method may not return at all; therefore, it would not work by naively labeling the return statements as method exit points. By recursively going through each execution path of the method, we observe and record the final return types of all program paths, and labels them as exit points. For method calls that the DSE engine has access to source code, we construct a VCFG for each of the method calls and collect all the exit-point locations. In the case of multiple nested method calls, the process goes on recursively until it gathers all required source code information. The final step is to connect the involved methods. In other words, it creates a complete graph that contains all possible branches in the program path. We create VCFG edges between all branching nodes that invoke a method call and the start nodes in the VCFG of the callee method. In this process, we also differentiate between methods that have the same name and yet have different package names or different parameters (overloading).

```
public int foo(int b)
{
    if (b % 2 == 1)
    {
        return 1;
    }
    return 0;
}

public int MultipleMethods(int a, int b)
{
    while(b < 100)
    {
        if(foo(b) == 1)
        {
            a--;
        }
        if(b < 0)
        {
            a++;
        }
        b++;
    }
    return a;
}
```

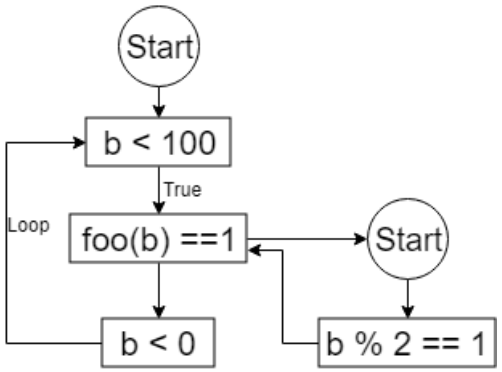

Figure 5.2: Example generated VCFG

5.1.4 Example of VCFG generation.

As a simple example, we illustrate our process for generating the VCFG for the code snippet in Figure 5.2. The `while (b < 100)` statement is determined as the first branch statement, and therefore we generate a VCFG node labeled with `b < 100`. After parsing this statement's enclosed statements, we generate a VCFG node labeled with `foo(b) == 1`. We draw a VCFG edge labeled with "True" between the two VCFG nodes because statement `if (foo(b) == 1)` is the first statement in `while (b < 100)`'s true-branch block. We then detect that the branch condition contains a method call to `foo`, and therefore proceed to generate the VCFG for method `foo`. In this case, the VCFG contains only the start VCFG node and the `b%2 == 1` VCFG node. We determine the `b%2==1` VCFG node to be an exit point, and therefore draw a VCFG edge between it and the `if(foo(b) == 1)` VCFG node to indicate the return relationship. Then, we determine the `if(foo(b) == 1)` statement as in a leaf position. When we create the `b < 0` VCFG node, we draw a VCFG edge with no label from the leaf `foo(b) == 1` VCFG node to it. Then, we determine `b < 0` as in a leaf position, and draw a VCFG edge with the "Loop" label from its VCFG node to the enclosing branch statement `foo(b) == 1`'s VCFG node. Finally, we send the generated VCFG to the exploration-data augmentor to obtain add-on statistics and information.

5.2 EXPLORATION-DATA AUGMENTOR

The exploration-data augmentor augments the data collected during the DSE exploration onto the VCFG created by the VCFG generator. In particular, the augmentor takes the VCFG and designated PUT written by the users as inputs, and it initiates the DSE exploration process to gather data at runtime. After every single run of DSE ends, the augmentor analyzes the result, and extracts the branching nodes on the execution path for aggregating information onto the VCFG nodes. For example, after a single DSE run of a method that contains two nested `if` statements, the augmentor extracts the information that the first `if` branch statement gets flipped once from "False" to "True", and therefore the augmentor increases the flip count by one in the corresponding VCFG node. At the last branching node of the execution path, the augmentor looks for possible error messages or generated test inputs, and appends them to the corresponding VCFG node. The path condition contained in each branching node gives hints on which part of the source code the execution node is located. The augmentor gathers the path conditions, and derives the incremental path condition based on the difference of path conditions between the current branching node and its parent branching node. Some branching nodes given by the DSE engine are implicit checks generated by the DSE engine or preconditions that do not exist in the source code. Therefore, the augmentor also identifies these branching nodes and exclude them from further processing.

5.2.1 Matching between VCFG Nodes and branching nodes in DSE

It is difficult to accurately fit the collected information onto the VCFG. The path-condition predicate from each branching node might differ from the actual branch condition in the source code. For example, if a branch is written as `if(x != 5)`, the path-condition predicate obtained from the DSE engine is `x == 5`. Furthermore, the augmentor needs information on whether the execution is stepping into the true-branch block or false-branch block of a branch statement so the augmentor can provide correct candidate VCFG node for the next branching node from DSE to append data to. Direct mapping by line numbers in the source code could address the issue, but unfortunately, such information is not always available or accurate in IntelliTest. Therefore, we choose two alternative techniques for checking whether a branch condition is *equivalent* to an incremental path condition, i.e., a path-condition predicate. The first technique is to extract the symbols and operators from the two conditions, and compare their syntactic differences. The second technique is to construct a truth table, and test the two predicates from the two conditions by instantiating

the symbols in the predicates with randomly sampled input data (note that this technique works for only primitive-type symbols such as integer and string).

5.2.2 Issues with variable name and value

If symbolic values are used as arguments for method invocation in branch conditions, it is likely that the corresponding parameters' names would be different in the caller method and the callee method. To track the correspondence, the parameter name of the callee method is saved onto a name stack. As the execution enters a new method or returns from a previous method, we push or pop the new name onto the name stack and keep track of it. It is also possible that a symbolic value changes (e.g., incremented) or becomes a concrete value in the execution because of constant-value assignment. For example, suppose `x` is a symbolic variable and there exists an increment statement on `x` such as `x++;`, in a later incremental path condition, this symbolic value can show up as `x+1`, which prevents us from mapping this incremental path condition to a correct VCFG node. Therefore, to address the issue, each symbolic variable is also tracked whether it is changed to a different value using a dictionary.

5.2.3 Collected information

This component collects the following information for each VCFG node. Note that the current VCFG node aggregates information from DSE execution paths along both true and false branches of the VCFG node's corresponding branch statement.

- Incremental path condition, being the predicate gathered from the branching nodes correspond to the current VCFG node. This information helps the users learn about the variety of the branching nodes. Note that the path condition is included in a different display location, i.e., in the pop-up window after clicking the node.
- Flip count, being the aggregated flip count of the constraint gathered from the incremental path conditions of the current VCFG node. This information helps the users learn the amount of effort that the DSE spends on the branch statement corresponding to the current VCFG node.
- Implicit checks. Some path conditions are not from the program but generated by the DSE-based tool such as nullity checking. We flag whether the incremental path condition is one of such implicit checks. Such information is not a displayed property.

- Run index. The DSE-based tool might run for multiple times on the same program, and each DSE run is uniquely indexed. The run index is a comma-separated list of all DSE runs that include the branch statement corresponding to the current VCFG node.

- Branch statement line number, being the line number of the branch statement corresponding to the current VCFG node. This information helps the users quickly find the corresponding source code if there is any confusion due to similar statements.

- Generated tests, being a comma-separated list of all generated test inputs at the current VCFG node.

- Reason of halting, being a comma-separated list of all the reasons of why the current execution path stops at the current VCFG node.

  We provide easy access of the preceding information to the users. Without such information, the users need to check both the IntelliTest output console and source code structure to dig out such information.

5.3 GRAPH VISUALIZER

The graph visualizer includes the visualization front-end to display the VCFG graph and information on it. In particular, to improve the guidance provided by the visualization result, we present an interactive graph with rich information to help the users. We use different colors and shapes to encode the information that the VCFG nodes contain and to help the users easily differentiate the different situations represented by the VCFG nodes. In the graph, each VCFG node represents one branch statement from the source code. We extract and use the Boolean predicate within the branch statement as the label for the VCFG node so that the users can quickly identify which line of code the branch statement belongs to. In case there are the same or similar branch conditions from the source code for multiple VCFG nodes, the users can click on each VCFG node to see the actual line number of the branch statement in the source code. Flip count is also shown on the VCFG node's label for convenience because it is an informative statistic in DSE exploration. The VCFG edges in the graph represent the execution flow of the program from one branch statement to the other. Self edges and back edges are possible as well to indicate loops. The arrows on the VCFG edges indicate the direction of the execution flow. It is possible to have two-way VCFG edges between VCFG nodes. According to the data gathered in the previous components,

the graph visualizer renders the information into color codes, text labels, and textual data in the pop-up window. During this process, we visualize the following information:

- Shapes:
  - A rectangle represents a VCFG node for a branch statement in the source code.
  - A circle represents a utility VCFG node, such as an entry point.
- Fill color:
  - White represents that the branch statement in the VCFG node does not contain symbolic variables in its branch condition. White indicates lower inspection priority.
  - Green represents that the branch statement does contain symbolic variables and has been reached at least once during the DSE exploration. Green is a safe color to indicate less threat to achieving code coverage.
  - Orange represents an un-flipped branch statement that contains symbolic variables. Orange is a warning color to indicate a threating factor.
  - Red represents an unreached branch statement during the DSE exploration. Red indicates a fatal error that should not appear.
  - Blue represents a utility VCFG node, such as an entry node, which is a node that does not come from the source code.

5.4 VISUALIZATION EXAMPLE

Listing 5.2: Method `Minus`

```
1   public int Minus(int xx)
2   {
3       string path = @"C:\mytest.txt";
4       string value = File.ReadAllText(path);
5       if (xx == 1)
6       {
7           for (int i = 0; i < value.Length; i++)
8           {
9               if (value[15] == 'c')
10                  xx++;
```

```
11                      if (i == -1)
12                      {
13                          return -1;
14                      }
15                  }
16                  if (xx == 150)
17                      return 0;
18              }
19              return 1;
20          }
```

Listing 5.3: PUT for method `Minus`

```
1           [PexMethod]
2           public int MinusTest([PexAssumeUnderTest]Program target,
3                   string value, int xx)
4           {
5               int result = target.Minus(xx);
6               return result;
7           }
```

For the `Minus` method in Listing 5.2, we present an example on how PexViz's visualization is helpful in determining the problem cause of the low code coverage.

After creating the PUT shown in Listing 5.3 and running IntelliTest, we find that the block coverage reported by the IntelliTest console is 1/12. From PexViz's visualization shown in Figure 5.3, the users can easily observe that the orange VCFG nodes are not flipped and one red VCFG node indicates unreached branch statement by the DSE exploration. By clicking on the red VCFG node, the users can observe that the DSE engine encounters a "FileNotFound" Exception from the "generated test" attribute. By looking at the red VCFG node's label, the users can easily observe that it corresponds to Line 5 in the source code. The users can infer that the exception comes from the lines related to file-system API calls before Line 5 since the red VCFG node indicates that Line 5 is never reached.

Listing 5.4: PUT with mock object for method `Minus`

```
1           [PexMethod]
2           public int MinusTest([PexAssumeUnderTest]Program target,
3                   string value, int xx)
4           {
```

```
5               using (ShimsContext.Create())
6               {
7                   System.IO.Fakes.ShimFile.ReadAllTextString = (x) =>
8                   {
9                       return value;
10                  };
11                  System.IO.Fakes.ShimFile.ExistsString = (x) => true;
12
13                  int result = target.Minus(xx);
14                  return result;
15              }
16          }
```

The problem can be solved by creating a mock object for the `File` object as shown in Listing 5.4. Running IntelliTest with PexViz again yields 10/12 block coverage. To investigate why the 2 remaining blocks are not covered, the users can look at the PexViz visualization shown in Figure 5.4. In the graph, majority of the VCFG nodes are in the green color, but one VCFG node is orange. As the orange color indicates that the branch statement does not get flipped, the possible cause for low block coverage could be from that branch statement. By inspecting the orange VCFG node, the users can trace back to the source code and discover that the condition is too strict to be flipped for branch statement `xx == 150`. The detailed reasons can be found in Chapter 7 for a similar problem. After adding a `PexAssume` statement such as `PexAssume.IsTrue(value[15] == 'c')` to the PUT, the DSE exploration is able to construct correct strings for variable `value` to flip the branch statement `xx == 150` and obtains 11/12 block coverage.

After further inspecting the visualization to find the cause for not covering the remaining 1 block, we find that the majority of the VCFG nodes have non-white colors and they all have similar flip counts. It can be inferred that the branch statements represented by these VCFG nodes have a high chance to be covered by the generated tests; therefore, the problematic not-covered block should come from the white VCFG node that represents the branch statements without symbolic value. By clicking on the white VCFG node and inspecting its source code, the users can find that the block can never be covered because `i` is always positive. The users can thus make the decision to omit it or modify the source code to correct the problem.

In summary, the PexViz approach provides the following benefits in the example:

- The colors and shapes of VCFG nodes help the users quickly identify un-flipped and unreached branch statements.
- The branch condition in the labels of VCFG nodes helps the users quickly identify the associated branch statement in the source code.
- The flip count statistic in the labels of VCFG nodes provides valuable and easily accessible statistic for understanding the DSE exploration.
- The "reason of halting" and other information embedded in the VCFG nodes help the users quickly associate the information with the associated branch statements.

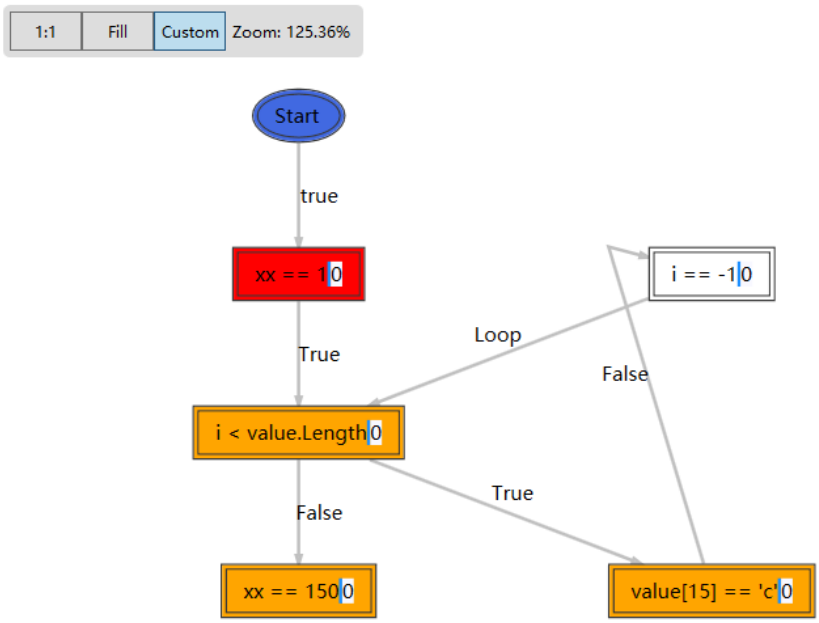

Figure 5.3: PexViz Visualization for method `Minus`

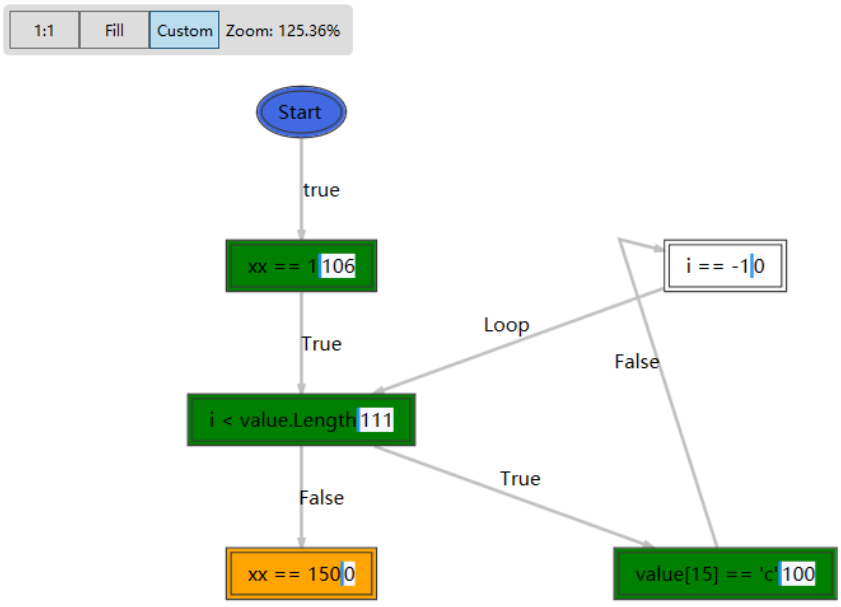

Figure 5.4: PexViz Visualization for method `Minus` with mock object

CHAPTER 6: IMPLEMENTATION

We implement our PexViz approach in C# as an extension to Visual Studio 2015. Our implementation includes three main components.

6.1 VCFG GENERATOR

The VCFG generator takes an input the target program's source code and outputs a customized QuickGraph object representing the control flow relationships in the program on a graph along with some metadata. The graph is serialized to the XML format and stored on disk. The implementation involves classic compiler techniques and utilizes static analysis techniques to analyze the variable dependency and function dependency. The actual implementation of the VCFG generator utilizes the .NET Compiler Platform SDK i.e., Roslyn[1] APIs for parsing source code.

6.2 EXPLORATION-DATA AUGMENTOR

The exploration-data augmentor first de-serializes the graph from the XML format file to a QuickGraph object. Then the exploration-data augmentor initiates the DSE process and hooks on different life cycles of the DSE processes (e.g., before each run, after each run, before exploration) to record information. The implementation of this exploration-data augmentor utilizes multiple data structures to record data such as Stack and Dictionary. The annotated graph is then serialized again to be stored in the XML format.

6.3 GRAPH VISUALIZER

The graph visualizer is implemented based on the Windows Presentation Foundation (WPF)[2] presentation system. The graphing capability relies on customized libraries of GraphSharp[3] and QuickGraph[4].

A customized visualization layout is defined in the XAML format. A C# component reads and de-serializes the graph from the XML format file on disk and populate the visualization

[1] `https://github.com/dotnet/roslyn`

[2] `https://docs.microsoft.com/en-us/dotnet/framework/wpf/getting-started/introduction-to-wpf-in-vs`

[3] `https://archive.codeplex.com/?p=graphsharp`

[4] `https://github.com/YaccConstructor/QuickGraph`

with the layout specified in the XAML file. The interactive components are action listeners written in C# to handle events such as zooming in and out, clicking and hovering with mouse. The customized generator for VCFG nodes is implemented to draw VCFG nodes according to the GraphSharp graph at runtime with the specified styles. Various layout algorithms are adapted and modified for better representation.

CHAPTER 7: CASE STUDY

This chapter presents a case study for problem diagnosis with our PexViz approach when running IntelliTest on two example code snippets, and diagnosing the boundary problem and object creation problem being faced, respectively. We also provide comparison with the default IntelliTest and SEViz being used for problem diagnosis on the same example snippets. The first code snippet is a modified bubble sort method from DSA[1] project. The second code snippet is from XUnit[2] project.

7.1 PROBLEM DIAGNOSIS FOR A BOUNDARY PROBLEM

For the code snippet of `BubbleSort` in Figure 7.1, we diagnose the faced boundary problem and try to improve block coverage achieved by IntelliTest. After creating a PUT for the code snippet and adding the SEViz and PexViz attribute properties to the test method for triggering SEViz and PexViz, respectively, we run IntelliTest exploration with default settings and obtain 11/14 block coverage. The IntelliTest console result indicates that 122 runs have completed until the engine stops because of reaching the timeout boundary.

Figures 7.2 and 7.3 show the interfaces of the IntelliTest console result and SEViz on the code snippet, respectively. The IntelliTest console result shows that IntelliTest achieves 11/14 block coverage, and generates 6 inputs that cover different blocks, along with 2 inputs that trigger exceptions. The warning message states that exploration ends due to reaching the boundary of "maximum runs without new tests". This particular information provides no further explanation of the fact that DSE fails to cover these 3 remaining blocks. In the graph generated by SEViz, in short as the SEViz graph, there are 276 nodes labeled with index numbers, much more than the 5 nodes in the graph generated by PexViz, in short as the PexViz graph. It is difficult for the users to navigate or digest the large amount of information efficiently from the SEViz graph to spot out useful information. To inspect different nodes in the SEViz graph, the users have to focus on the pop-up window to look for the incremental path condition and guess which branch in the source code that a node corresponds to. It is difficult for the users to associate different subtrees together because the users do not have direct information on what the nodes are about and the relationships between different nodes. However, by clicking through a large number of starting nodes in the subtrees with similar shapes, we observe that the incremental path conditions are similar

[1] `https://archive.codeplex.com/?p=dsa`
[2] `https://github.com/xunit/xunit`

```
13          public void BubbleSort(int[] number, int n)
14          {
15              int count = 0;
16              for (int i = 0; i < number.Length;i++)
17              {
18                  count++;
19                  for (int j = 0;j < n;j++)
20                  {
21                      if (number[i] > n)
22                      {
23                          int temp = number[j];
24                          number[j] = number[j + 1];
25                          number[j + 1] = temp;
26                      }
27                      if(count == 5)
28                      {
29                          count = 0;
30                      }
31                  }
32              }
33
34              if (number.Length > 200)
35              {
36                 throw new Exception("bug");
37              }
38          }
39      }
40  }
```

Figure 7.1: A code snippet of `BubbleSort` where a boundary problem is faced

for these nodes. Consequently, we suspect that the branch statement associated with such nodes may be related to the undesired coverage situation, but we could not infer further meaningful actionable results.

Next we investigate the PexViz graph as shown in Figure 7.4. There are 6 VCFG nodes in the PexViz graph, which has 97.8% fewer nodes than the 276 nodes from the SEViz graph for the users to inspect. We start examining the PexViz graph from the blue entry VCFG node. We can directly observe the clear correspondence between VCFG nodes and branch statements through the VCFG node labels. Thus, such mechanism saves us navigation time in contrast to clicking through each of the 276 nodes in the SEViz graph. The orange VCFG node showing 0 flip count immediately draws our attention with distinct color compared to all other VCFG nodes' colors. According to the orange VCFG node's label, the condition is expected to be evaluated to "True" if the length of the `number` array is larger than 200. To gain further understanding of the reason why the branch is not flipped, we examine the three neighboring green VCFG nodes. All three green VCFG nodes have flip count of around 120.

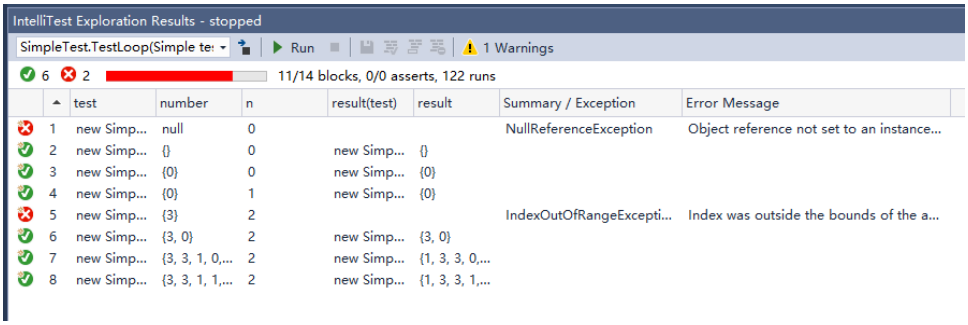

Figure 7.2: IntelliTest console result on running IntelliTest against `BubbleSort`

The information on the generated test inputs that we observe after clicking on the three green VCFG nodes indicates that the array with length larger than 200 is not created. The DSE process stops before it is able to generate an array with length 200; therefore, increasing the bound of the maximum number of runs can be a solution to the problem. After we increase the maximum-run bound, IntelliTest manages to reach 14/14 (100%) block coverage.

7.2 PROBLEM DIAGNOSIS FOR AN OBJECT CREATION PROBLEM

We create a PUT for the code snippet of `GetMessage` in Figure 7.5 and run IntelliTest with SEViz and PexViz, respectively. Based on the information in the IntelliTest console result, the exploration stops after 3 runs and achieves 11/21 block coverage, and there are no warnings.

Figures 7.6 and 7.7 show the interfaces of the IntelliTest console result and SEViz on the code snippet, respectively. The IntelliTest console result shows that the tool achieves 11/21 block coverage with 1 generated input, and there is no warning message. The information is quite limited to help the users figure out where and what the problem is for causing DSE not to cover the remaining 10 blocks. In the SEViz graph, there are 8 nodes. After we click and activate the nodes on the left, the pop-up windows show no useful information other than the generated test input in the green node. There is no source-code mapping or path condition recorded for the nodes. For the orange node on the left, we can see that the incremental path condition is `msg == (Exception)null` but it still gives no insight on why and which branches or statements are not covered.

In contrast, the PexVix graph shown in Figure 7.8 includes 5 VCFG nodes connected by a straight line. Two white VCFG nodes do not contain symbolic values; therefore, they have lower priority to be inspected. The two orange VCFG nodes that have symbolic values in the branch condition have 0 flip count. We notice that both branches of the orange VCFG nodes

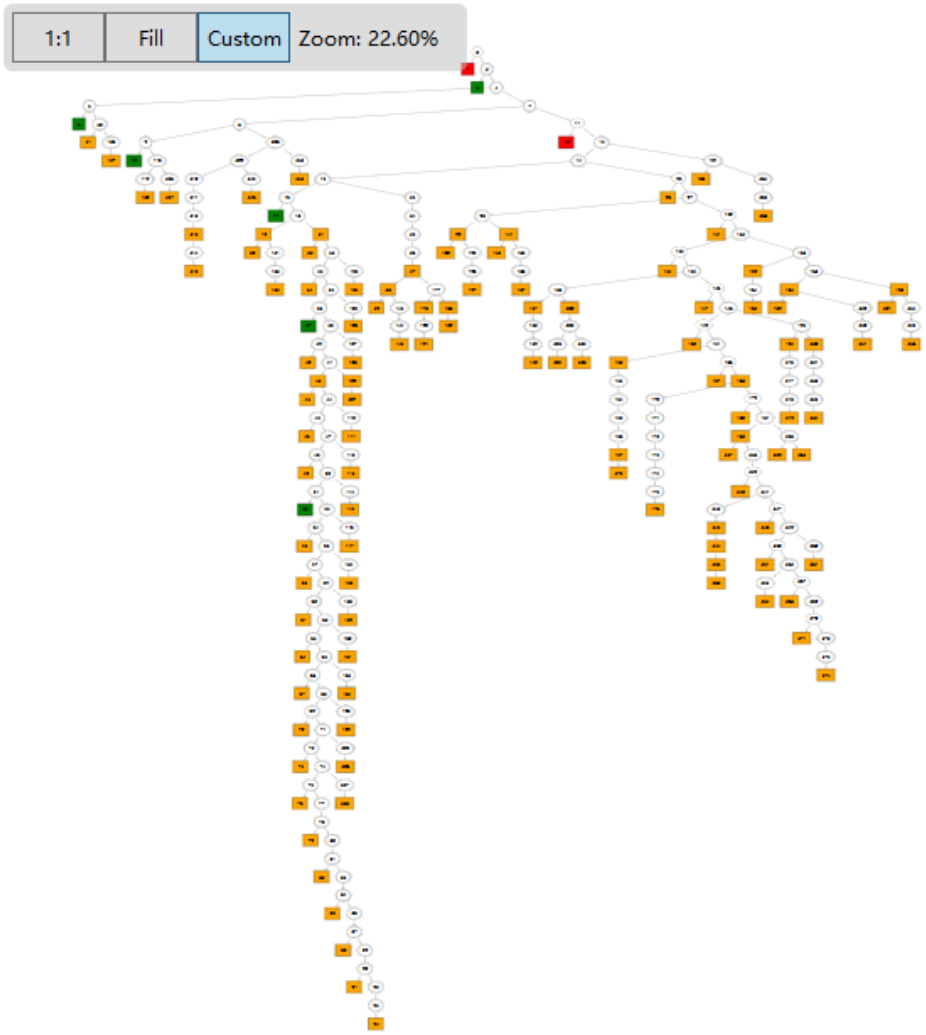

Figure 7.3: SEViz visualization on running IntelliTest against `BubbleSort`

relate to some properties of the symbolic variable `ex` such as its object value and object-field values. By going back to the source code, we observe the type of symbolic variable `ex` is not primitive but is of an `Exception` class type. Looking at the not-flipped branches again together with the inputs generated by IntelliTest, we find that the guessed object `ex` does not have the states to satisfy the negation of the constraints from these branches. With such information, we construct a factory method to synthesize an `Exception` object that can have varied states, and then running IntelliTest again yields 21/21 (100%) block coverage.

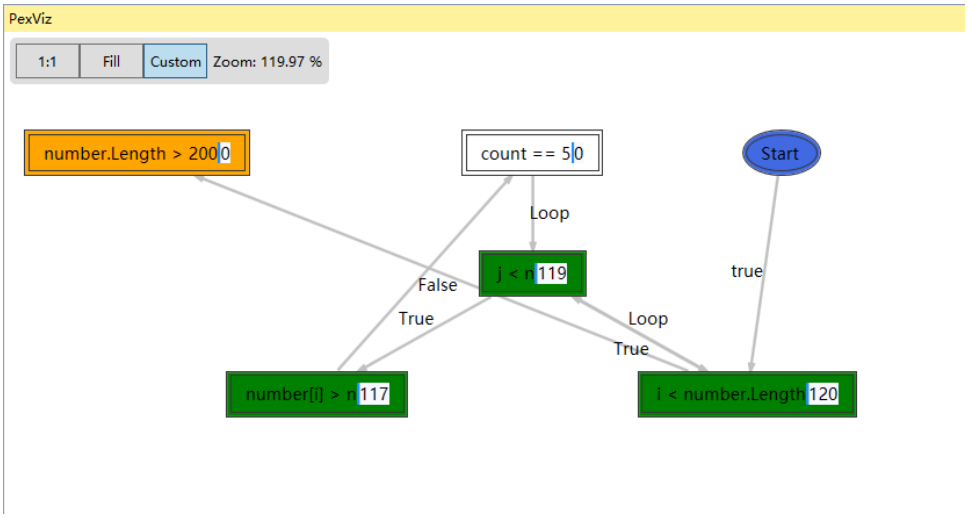

Figure 7.4: PexViz visualization on on running IntelliTest against `BubbleSort`

```
53  public static class ExceptionUtility
54  {
55      const string RETHROW_MARKER = "$$RethrowMarker$$";
56
57
        2 references | jcao7, 4 days ago | 2 authors, 3 changes
58      public static string GetMessage(Exception ex, int level)
59      {
60          string result = "";
61
62          if (level > 0)
63          {
64              for (int idx = 0; idx<level; idx++)
65                  result += "----";
66
67              result += " ";
68
69          }
70
71          if ( !( ex is AssertException ) )
72             result += ex.GetType().FullName + " : ";
73
74          result += ex.Message;
75
76          if (ex.InnerException != null)
77              result = result + Environment.NewLine + GetMessage(ex.InnerException, level + 1);
78
79          return result;
80      }
81
```

Figure 7.5: A code snippet of `GetMessage` where an object creation problem is faced

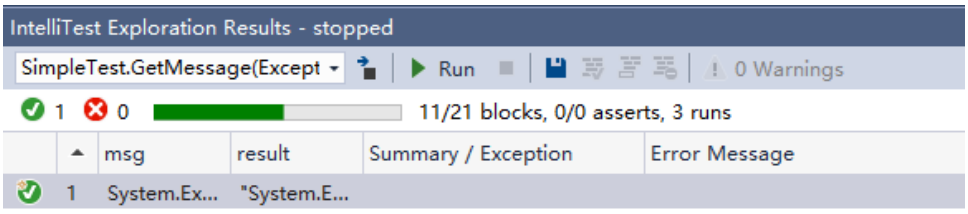

Figure 7.6: IntelliTest console result on running IntelliTest against `GetMessage`

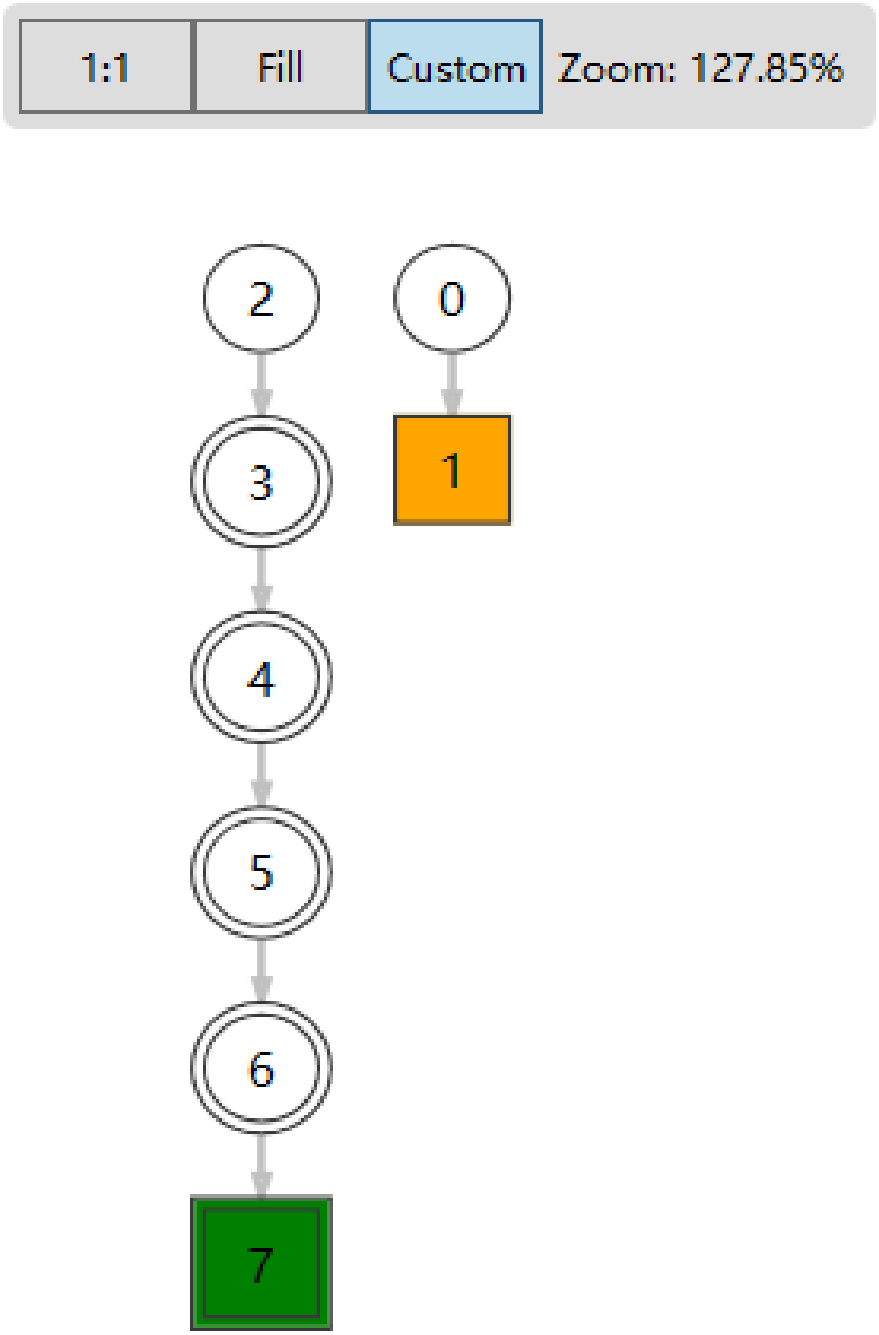

Figure 7.7: SEViz visualization on running IntelliTest against `GetMessage`

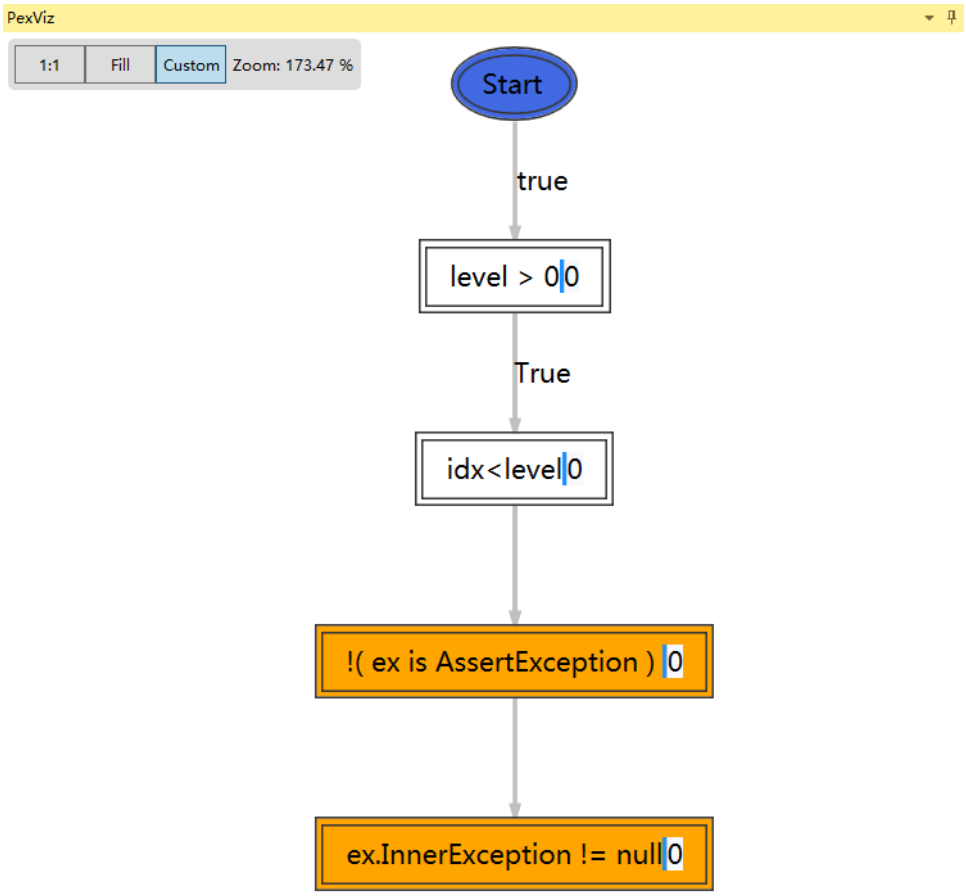

Figure 7.8: PexViz visualization on on running IntelliTest against `GetMessage`

CHAPTER 8: DISCUSSION

This chapter discusses the limitations of the current PexViz approach and implementation along with our planned future work to address such limitations.

8.1 COLOR SCHEMA

The color schema for VCFG nodes with different situations is currently a fixed setting. Five different colors represent five different situations but the meanings of the assigned colors do not follow a uniform or comprehensible standard. In future work, we plan to use a different color space and wider color range to provide users richer information. For example, in order to make the different color dimensions more sensible to human vision, we plan to use the HSV color space [36], with the planned color schema as follows:

- H (Hue, range: 0-360): $=(FlipCount/MaximumFlipCount) * 360$
- V (Value/Brightness, range: 0-100):$\frac{NumberOfErrorsAndWarnings}{TotalNumerOfWarnings} * 100$

8.2 INFORMATION ASSOCIATED WITH VCFG EDGE

In our current approach, there is no direct visual information about whether a branching node is flipped from the false branch to the true branch or from the true branch to the false branch. Currently the displayed flip count associated with a VCFG node is the sum of the number of flippings from the false branch to the true branch and the number of flippings from the true branch to the false branch. Such detailed information differentiating the flipping direction can help tool users identify whether DSE exploration spends sufficient resources on flipping the "most rewarding" branching nodes. To provide such detailed information to the tool users, our Data Augmentor can gather it during the exploration runtime, and then our Visualizer can visualize the thickness of a VCFG edge according to the number of times such edge is explored/traversed.

We next use an example to illustrate the preceding limitation and planned work to address it. For the code snippet in Figure 8.1, steps are taken to write a PUT and then run IntelliTest with PexViz on the PUT. The IntelliTest console shows 10/11 block coverage reached and the process stops after 109 runs due to reaching the default setting "Max runs without new tests" of 100. At the first glance, the visualization in Figure 8.2 shows a similar layout as the example boundary problem in the case study (Chapter 7). The VCFG node `x == 300` is

```
202         public bool TestLoop3(int x, int[] y)
203         {
204              if (x == 90)
205             {
206                 for (int i = 0; i < y.Length; i++)
207                 {
208                     if ( y[i] == 15 )
209                     {
210                         x++;
211                     }
212                 }
213                 if (x == 300)
214                      return true;
215             }
216              return false;
217         }
```

Figure 8.1: A code snippet of `TestLoop3` where a boundary problem is faced

colored with orange, and x is not large enough to satisfy the constraint from the conditional. However, increasing the threshold of max runs does not help increase the coverage ratio. By carefully examining the source code, we notice that in order to increase `x`, `y[i]` has to be equal to 15 for at least 300 minus 90 times. From the current visualization, there is not much information to provide more insights; however, if the VCFG edges are visualized with thickness, we can easily notice the flipping on the VCFG node `y[i] == 15` mostly happens from the false-to-true direction, indicating that `y[i]` is initialized with a non-15 value (in fact being 0, the default value for the integer type). In the end, the number of loop iterations (reflecting the number of `y`-array elements) cannot be expanded to be equal or greater than 210. Note that expanding the loop (increasing the array length) is through the flipping of the loop condition `i < y.Length` (in the last loop iteration) from false to true. Thus, the true branch of `x == 300` (in the orange node) is never covered after the end of the DSE exploration. A solution to cover the true branch of `x == 300` after knowing such information is to add an assumption to specify the length of `y` to be greater than 210 and/or the values in the array to be 15.

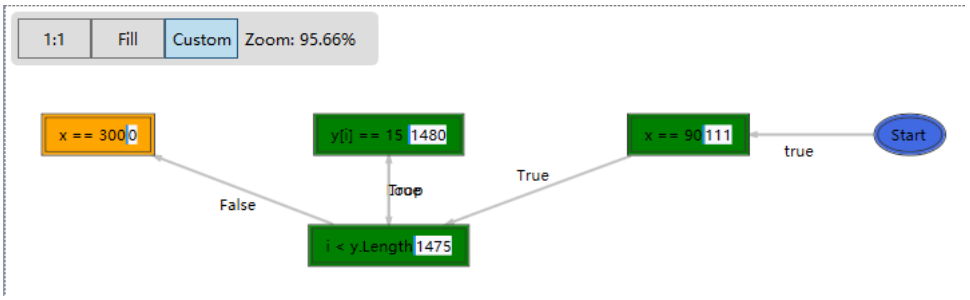

Figure 8.2: PexViz visualization on the boundary problem faced in the code snippet of `TestLoop3`

CHAPTER 9: CONCLUSION

In this thesis, we have presented a visualization approach named PexViz that helps users of a tool based on Dynamic Symbolic Execution (DSE) better understand and diagnose the encountered problems. PexViz reduces the large search space for problem root causes by aggregating and visualizing information gathered through DSE exploration. We have presented the motivations for our approach by investigating the limitations of previous approaches, and illustrated our design decisions for the visualization in our approach. We have provided a comparison between our approach and two related previous approaches (IntelliTest and SEViz). Our initial comparison results on example cases have demonstrated the benefits of our PexViz approach and its superiority over the related approaches. In future work, we plan to conduct a user study to further assess the usability of our PexViz approach to assist tool users for diagnosing the faced problems when applying a DSE-based tool.

REFERENCES

[1] X. Xiao, S. Thummalapenta, and T. Xie, "Advances on improving automation in developer testing," in *Advances in Computers*, 2012, vol. 85, pp. 165–212.

[2] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, p. 10, 2008.

[3] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, 2005, pp. 213–223.

[4] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2005)*, 2005, pp. 263–272.

[5] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems & Networks (DSN 2009)*, 2009, pp. 359–368.

[6] X. Xiao, T. Xie, N. Tillmann, and J. De Halleux, "Covana: Precise identification of problems in Pex," in *Proceedings of 33rd International Conference on Software Engineering (ICSE 2011)*, 2011, pp. 1004–1006.

[7] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux, "Precise identification of problems for structural test generation," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, 2011, pp. 611–620.

[8] T. Xie, L. Zhang, X. Xiao, Y. Xiong, and D. Hao, "Cooperative software testing and analysis: Advances and challenges," *J. Comput. Sci. Technol.*, vol. 29, no. 4, pp. 713–723, 2014.

[9] N. Tillmann and J. De Halleux, "Pex – white box test generation for .NET," in *Proceedings of International Conference on Tests and Proofs (TAP 2008)*, 2008, pp. 134–153.

[10] T. Xie, "Transferring software testing tools to practice," in *Proceedings of 2017 International Workshop on Automation of Software Testing (AST 2017)*, 2017, pp. 8–8.

[11] D. Honfi, A. Voros, and Z. Micskei, "SEViz: A tool for visualizing symbolic execution," in *Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*, 2015, pp. 1–8.

[12] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, pp. 385–394, 1976.

[13] M. Lin, Y. Chen, K. Yu, and G. S. Wu, "Lazy symbolic execution for test data generation," *IET software*, vol. 5, no. 2, pp. 132–141, 2011.

[14] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on software engineering*, no. 3, pp. 215–222, 1976.

[15] J. De Halleux and N. Tillmann, "Parameterized unit testing with Pex," in *Proceedings of International Conference on Tests and Proofs (TAP 2008)*, 2008, pp. 171–181.

[16] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," in *Proceedings of 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, 2005, pp. 365–381.

[17] S. Thummalapenta, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte, "MSeqGen: Object-oriented unit-test generation via mining source code," in *Proceedings of 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009)*, 2009, pp. 193–202.

[18] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su, "Synthesizing method sequences for high-coverage testing," in *Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2011)*, 2011, pp. 189–206.

[19] D. Thomas and A. Hunt, "Mock objects," *IEEE Software*, vol. 19, no. 3, pp. 22–24, 2002.

[20] D. Saff and M. D. Ernst, "Mock object creation for test factoring," in *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2004)*, 2004, pp. 49–51.

[21] T. Mackinnon, S. Freeman, and P. Craig, "Endo-testing: unit testing with mock objects," *Extreme programming examined*, pp. 287–301, 2000.

[22] M. R. Marri, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "An empirical study of testing file-system-dependent software with mock objects," in *Proceedings of 4th International Workshop on Automation of Software Test (AST 2009), Business and Industry Case Studies*, 2009, pp. 149–153.

[23] K. Taneja, Y. Zhang, and T. Xie, "MODA: Automated test generation for database applications via mock objects," in *Proceedings of 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, 2010, pp. 289–292.

[24] L. Zhang, X. Ma, J. Lu, N. Tillmann, J. de Halleux, and T. Xie, "Environment modeling for automated testing of cloud applications," *IEEE Software, Special Issue on Software Engineering for Cloud Computing*, vol. 29, no. 2, pp. 30–35, 2012.

[25] X. Xiao, S. Li, T. Xie, and N. Tillmann, "Characteristic studies of loop problems for structural test generation via symbolic execution," in *Proceedings of the 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE 2013)*, 2013, pp. 246–256.

[26] N. Tillmann and W. Schulte, "Parameterized unit tests," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2005)*, 2005, pp. 253–262.

[27] S. Thummalapenta, M. Marri, T. Xie, N. Tillmann, and J. de Halleux, "Retrofitting unit tests for parameterized unit testing," in *Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE 2011)*, 2011, pp. 294–309.

[28] W. Lam, S. Srisakaokul, B. Bassett, P. Mahdian, T. Xie, P. Lakshman, and J. de Halleux, "A characteristic study of parameterized unit tests in .NET open source projects," in *Proceedings of the 32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, 2018.

[29] F. E. Allen, "Control flow analysis," in *Proceedings of the Symposium on Compiler Optimization*, 1970, pp. 1–19.

[30] D. H. et al, "SEViz," https://github.com/FTSRG/seviz, 2015.

[31] Y. Song, "More usable recommendation systems for improving software quality," Ph.D. dissertation, Department of Computer Science, North Carolina State University, 2014.

[32] W. De Pauw and J. Vlissides, "Visualizing object-oriented programs with Jinsight," in *Proceedings of European Conference on Object-Oriented Programming (ECOOP 1998)*, 1998, pp. 541–542.

[33] R. Hähnle, M. Baum, R. Bubel, and M. Rothe, "A visual interactive debugger based on symbolic execution," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, 2010, pp. 143–146.

[34] M. Hentschel, R. Hähnle, and R. Bubel, "Visualizing unbounded symbolic execution," in *Proceedings of International Conference on Tests and Proofs (TAP 2014)*, 2014, pp. 82–98.

[35] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, 2002, pp. 467–477.

[36] A. R. Smith, "Color gamut transform pairs," in *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1978)*, 1978, pp. 12–19.

[37] M. Recheis, "GitSharp," https://github.com/henon/GitSharp, 2018.

[38] D. Moody, "The physics of notations: toward a scientific basis for constructing visual notations in software engineering," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 756–779, 2009.