SEViz: A Tool for Visualizing Symbolic Execution

Dávid Honfi, András Vörös and Zoltán Micskei
Budapest University of Technology and Economics
Budapest, Hungary
Email: david.honfi@inf.mit.bme.hu, vori@mit.bme.hu, micskeiz@mit.bme.hu

Abstract—**Generating test inputs from source code is a topic that is starting to transfer from academic research to industrial application. Symbolic execution is one of the promising techniques for such white-box test generation. However, test input generation for non-trivial programs often reaches limitations even when using the most mature tools due to the underlying complexity of the problem. In such cases, visualizing the symbolic execution and the test generation process could help to quickly identify required configurations and modifications that enable the generation of further test inputs and increase coverage. We present a tool that is able to interactively visualize symbolic execution. We also show how this tool can be used for educational and engineering purposes.**

I. Introduction

Nowadays, software testing trends show that automated test generation is spreading amongst industrial software development projects. Several methods have been proposed to automatically generate test inputs from source code [1], one of them being *symbolic execution*. Symbolic execution is a program analysis technique, where the possible paths of the code are represented by constraint expressions over symbolic variables. By solving these constraints, input values can be gained that cover the different execution paths of the program.

This technique was first introduced in the 80s [2], and numerous enhancements had been proposed since then [3], [4]. Symbolic execution starts from a predefined entry point of the program, and then interprets each statement in parallel with gathering the constraints. A path in a symbolic execution trace can be mapped to an exact execution cases of the program. Every path has its condition (PC—Path condition), which is typically a quantifier free FOL formula interpreted as an SMT problem [5], containing the expressions over the variables discovered during the execution. Solving a formula of a path results in concrete input values driving the program exactly through this path. A typical use case of symbolic execution is generating inputs for white-box tests achieving high code coverage.

Microsoft Pex [6] is a white-box test generation tool integrated in Visual Studio, which uses dynamic symbolic execution, an enhanced variant that combines symbolic execution with concrete executions.

Industrial adoption is a current research topic of symbolic execution, because there are many hindering factors to come across (e.g. precise problem identification, efficient usage on large-scale software) [7]–[10]. Our experiences confirmed the challenges reported by others. We used Microsoft Pex in testing a model checker tool developed at our research group and a content management software from one of our local industrial partners[1]. In similarly complex software the first executions typically achieve low coverage and miss important test inputs. This is usually due to reaching timeouts or code accessing external libraries and resources that cannot be represented, or conditions that cannot be solved efficiently. The test input generator tool reports these issues, but identifying and correcting the root cause of them may prove to be effort-intensive, and may involve analyzing the detailed logs of the tool. Moreover, this could be a non-trivial task for developers and test engineers without a strong academic background.

Note that the above issue is not specific to Microsoft Pex, which is one of the most mature white-box test generation tools, these are the consequences of the theoretical limitations and challenges of the symbolic execution-based approach.

To overcome the gap discovered in industrial applications, we applied the idea of visualizing symbolic execution. SEViz (*Symbolic Execution VIsualiZer*), the prototype tool introduced in this paper, interactively visualizes symbolic executions in a form of symbolic execution trees. The nodes in this tree provide information about the test generation (e.g. path conditions, source code locations, selected test inputs). The visualization serves as a quick overview of the whole execution and helps to enhance the test input generation. The tool is capable of collecting the data to be visualized by monitoring the execution of Pex, and is able to communicate with Visual Studio in order to have an interactive visualization with source code inspection.

Our initial motivation for developing SEViz was to assist testing complex software, however, the tool can also be used in education and training by showing the process and result of symbolic execution in a step-by-step manner on simpler programs.

The paper summarizes our experiences for i) identifying what to visualize in terms of symbolic execution and how to represent each element, ii) showing how visualization can enhance test input generation, thus the whole symbolic execution process, and iii) implementing the components of SEViz.

SEViz can be dowloaded from the following site:

`http://ftsrg.github.io/seviz`

[1]Note, these case studies we performed independently from Microsoft and the Pex team, and were not published.

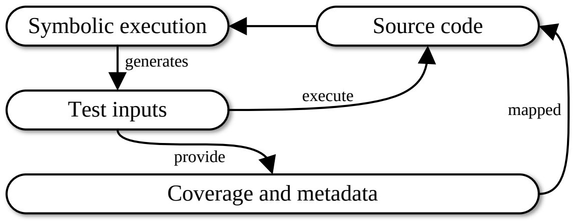
In Proc. of IEEE Int. Conf. on Software Testing, Verification and Validation (ICST) 2015, Tool Track. DOI: 10.1109/ICST.2015.7102631

The paper is organized as follows. Section II presents a motivational example. Section III describes the design decisions of the visual representation. Section IV illustrates how the tool can be used for educational or advanced test generation purposes. Section V details the lessons learnt during implementation. Section VI overviews the related work and Section VII concludes the paper.

II. Motivation

This section introduces an example, which shows the typical issues encountered in generating test inputs for complex software. Fig. 1 illustrates the symbolic execution-based test generation approach used in the example. Our goal in this example is to select test inputs that could identify possible failures and achieve relatively high code coverage.

Fig. 1. An approach for using symbolic execution



We selected one method (`Resolve`) from the GitSharp open source project [11], which is a real-world, .NET-based Git client application. The method parses so-called Git revision strings (a unique identifier of a modification set) and returns the identifier of the corresponding object found in the Git repository. The method `Resolve` is one of the most complex parts in the GitSharp project with its 147 lines of code. It has 28 `if` statements, 8 `while` and 7 `for` loops, and 2 `switch` statements. The cyclomatic complexity of the method is 51.

The description of the example uses these concepts from the terminology of the Pex tool. *Execution* refers to the whole symbolic execution process, and *run* is an execution of one, exact path. *Precondition*s provide hints to the symbolic execution engine for selecting the most interesting inputs (e.g. a parameter should be not null or has a specific format). *Mock*s help to isolate the unit under test from its dependencies. Pex uses so called *boundaries* to limit symbolic execution (e.g. number of method calls).

a) Basic test generation: First, Pex was started with the default configuration and without any guidance or intervention, so there were no preconditions or mock objects to restrict the symbolic execution. This execution produced a block coverage of 22.15% and generated 12 test inputs. The tool reached a specific boundary during execution (reached 100 runs without a new test), and warned about file system access in the code (Pex does not instrument the file access API by design). Increasing the limit for the boundary or increasing the time limit for the execution did not result in generating new test input.

b) Isolated test generation: In the next step, as suggested by the tool, we used mock objects to isolate the file system. After re-executing Pex the results showed that the file system was successfully isolated, but the boundary was still reached, thus code coverage was not increased significantly.

This is the critical point of the example, as we have to find out the cause of the unsuccessful test generation, and make the necessary adjustments (e.g. reconfigure the tool or make the unit under test more testable) to achieve additional generated test inputs and higher code coverage. This is a typical situation we observed, where several questions arise:

- *What kind of information can help in the analysis?* Several factors affect the success of the test generation, e.g. the conditions in the code, the search strategy used by the symbolic execution engine.
- *Where to find this information?* The basic information is displayed on the UI of the tool, but most of it is in the detailed reports or in the debug logs.

c) Test generation with report analysis: Finally, we thoroughly analyzed the data in the report generated by Pex manually, which guided us to discover, that Pex could not guess the correct form of the revision string, and that is the reason why it could not cover new statements and branches. Based on this, we specified test input preconditions that show the specific format of the revision string (e.g. it should contain at least one of the following strings: `^`, `@`, `~`, `tree`, `commit`). This execution reached 50.63% block coverage and generated 21 more test inputs, a significant increase compared to the basic step.

Even though we were already experienced in symbolic execution-based test generation, the initial analysis of the logs and reports was quite effort-intensive (requiring 1-2 hours just for this method). The time required for this analysis can be greatly reduced if the necessary information is collected in a clear representation, which helps quickly understanding the execution and makes searching for the required details fast and easy. The requirements against such representation are the followings.

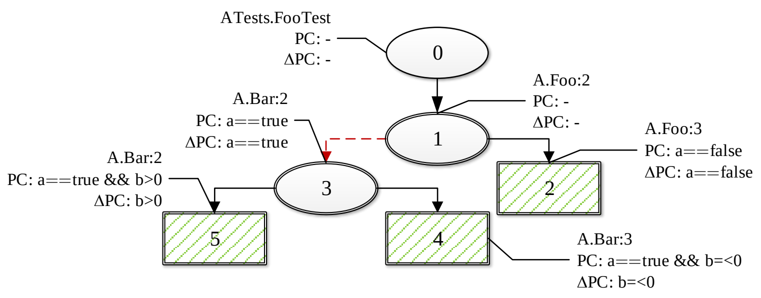
1) *Clarity:* It should be clear and easily understandable for users, who are just getting to know symbolic execution.
2) *Traceability:* It should be connected with source code and traceable in order to discover where did the execution end exactly.
3) *Detailed:* It should be annotated with detailed information about the symbolic execution (e.g. path conditions).

The next section will describe our chosen visual representation.

III. Visualization

Based on the previous functional requirements, program execution traces can be represented as a directed graph with metadata attached to each of its nodes. This directed graph is the symbolic execution tree. Each node represents a location in the explored code. An edge from one to another node exists if the statements were interpreted after each other in one

Fig. 2. Example symbolic execution tree with metadata



execution. Since directed graphs are well-known, we can state that, this fulfills the clarity requirement of the representation.

A. Information

Based on our experiences with test input generation we chose the following metadata to appear in each *node* of the symbolic execution tree.

Basic Information:

- *Sequence number:* Represents the order of the given node in the symbolic execution.
- *Method:* Each node points to the containing method, where the corresponding source code statement is located.
- *Source code (if exists):* Mapping a node to a source code location can also give a considerable help for overview, thus nodes should be mapped to it, if the source code is available. If not, the method of the node could help.
- *Tracing runs:* Each run should be individually traceable, which helps discovering, where the given run possibly ended or stranded.

Detailed Information:

- *Path condition:* The constraints, which have been collected until reaching the examined node.
- *Constraint solver calls:* By marking when a call to the constraint solver is made, issues related to constraint solver can be discovered.
- *Generated tests:* Shows whether a test was generated in a leaf node or not.
- *Reason of exhaust:* The information on the state of the node after the execution. This data can tell if there is a chance of discovering a new path branching from this node (e.g. not yet exhausted or timeout), if not the reason is provided (e.g. it is a leaf).

In large-scale software, path conditions can grow into large constraint systems, which can be hard to understand. To overcome this issue, we present the path conditions also in an *incremental* format attached to each node. This shows which literal has been added to the constraint system compared to the previous node. It could help locate where that constraint was added that can not be handled by the constraint solver.

B. Visual Representation

Designing a consistent, intuitive visual representation is not straightforward [12]. After several iteration we selected that the shape, border and color of the nodes should represent some of the basic information mentioned above, in the following way.

- *Shape:* If the constraint solver was called during executing the node, the shape of the node is an ellipse, otherwise it is a rectangle.
- *Border:* If source code mapping is available for the node, its border is doubled, otherwise it is simple-lined.
- *Fill color:* Each leaf node is colored depending on its role. If the execution was ended without a newly discovered path, then the fill color should be orange. Otherwise, if the execution ended without any error or exception, the node should be green, else it is red.

Edges can also display metadata. Our idea is to mark the edges with red color, if the symbolic execution is leaving a predefined unit of the source code. Unit isolation can be utilized here. Otherwise, the edges stay black.

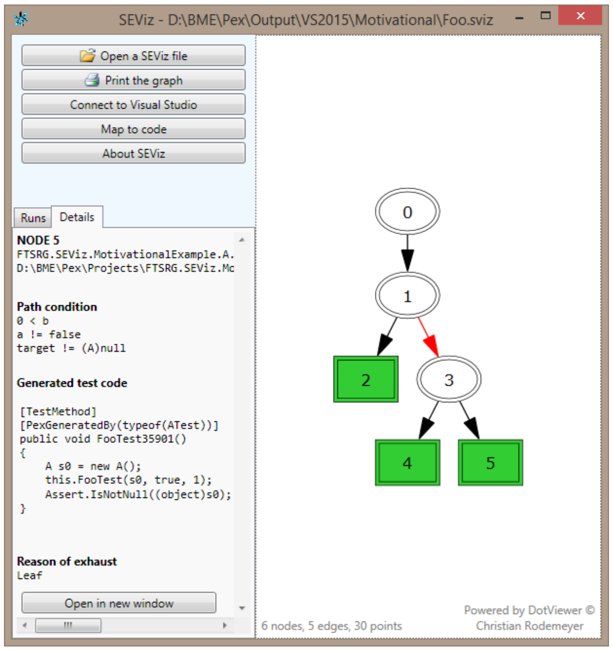
C. Visualization Example

Let us consider the following example, in which the source code below has its symbolic execution tree visualized. Method `Foo` of class `A` is our unit under test for now.

```
class A
{
    int Foo(bool a, int b)
    {
        if(a) return Bar(b);
        else return -1;
    }
    int Bar(int b)
    {
        if(b > 0) return 1;
        else return 0;
    }
}
```

If we visually represent the symbolic execution starting from method `Foo`, the symbolic execution tree with extended metadata will be as shown in Fig 2. The nodes are annotated with detailed information discussed above. The red dashed edge indicates that the execution exits from the examined unit.

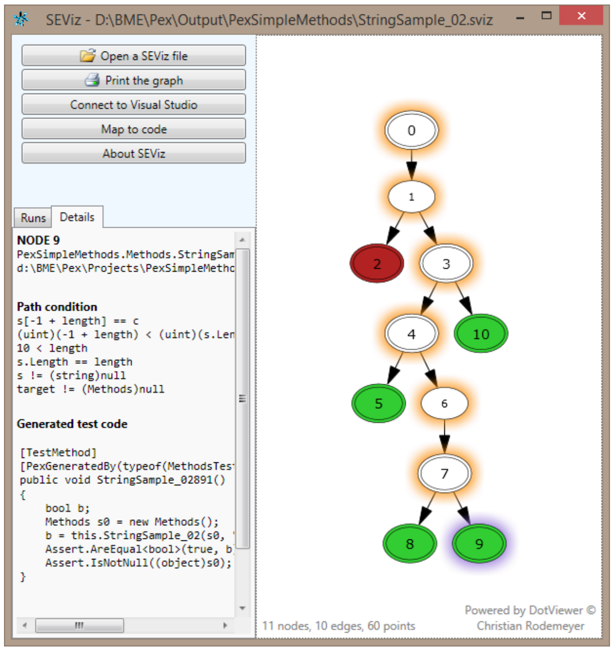
Fig. 3. Symbolic execution trees in SEVIZ



SEViz - D:\BME\Pex\Output\VS2015\Motivational\Foo.sviz
Open a SEViz file
Print the graph
Connect to Visual Studio
Map to code
About SEViz
Runs
Details
Open in new window
0
1
2
3
4
5
6 nodes, 5 edges, 30 points
Powered by DotViewer © Christian Rodemeyer

```
NODE 5
FTSRG.SEViz.MotivationalExample.A.
D:\BME\Pex\Projects\FTSRG.SEViz.Mc

Path condition
0 < b
a != false
target != (A)null

Generated test code

 [TestMethod]
 [PexGeneratedBy(typeof(ATest))]
 public void FooTest35901()
 {
     A s0 = new A();
     this.FooTest(s0, true, 1);
     Assert.IsNotNull((object)s0);
 }

Reason of exhaust
Leaf
```

(a) Visualization example



SEViz - D:\BME\Pex\Output\PexSimpleMethods\StringSample_02.sviz
Open a SEViz file
Print the graph
Connect to Visual Studio
Map to code
About SEViz
Runs
Details
0
1
2
3
4
10
5
6
7
8
9
11 nodes, 10 edges, 60 points
Powered by DotViewer © Christian Rodemeyer

```
NODE 9
PexSimpleMethods.Methods.StringSam
d:\BME\Pex\Projects\PexSimpleMetho

Path condition
s[-1 + length] == c
(uint)(-1 + length) < (uint)(s.Len
10 < length
s.Length == length
s != (string)null
target != (Methods)null

Generated test code

 [TestMethod]
 [PexGeneratedBy(typeof(MethodsTes
 public void StringSample_02891()
 {
     bool b;
     Methods s0 = new Methods();
     b = this.StringSample_02(s0, '
     Assert.AreEqual<bool>(true, b
     Assert.IsNotNull((object)s0);
 }
```

(b) String example

IV. USE CASES

This section presents the two main use cases of our tool. First, by visualizing the symbolic execution of simple programs with basic language elements (branches, loops, etc.), users can get familiar with the technique. Secondly, it is able help test engineers and developers to make the correct decisions to enhance test generation.

The typical usage workflow of SEVIZ is rather straightforward. Currently, it collaborates with Microsoft Pex, therefore we introduce the workflow using it.

1) *Attributing:* The user should specify two attributes for the unit test method in order to start monitoring during the execution of Pex. These attributes define the output file location and the unit under test.
2) *Execution:* During the execution of Microsoft Pex, the monitoring is active and data is collected.
3) *Analysis:* When the execution finishes, SEVIZ creates a file, which contains all the collected information. This can be opened with the tool and a symbolic execution tree is drawn from the data.
4) *Mapping:* Each node, which has source code metadata, can be mapped to a source code file and line in a user specified Visual Studio instance.

A. Educational and Training Usage

The motivation of our work was to support engineers using symbolic execution for test generation in large-scale, complex software cases. However, SEVIZ, the implemented tool has another use case, which is to support the learning curve of the usage of symbolic execution. From simple to rather complex cases, our tool is able to visualize the executions, therefore users, who just started to learn symbolic execution, can have the desired knowledge with less effort.

Users can easily extract the path conditions and observe the executions as the contained constraints are incrementing with literals from node to node (by using the incremental path conditions). Especially, cases with complex objects are challenging, since the path condition constructs are can grow into rather complex systems.

Source code mapping and the overall view of the symbolic execution tree can help understanding and comparing the search strategy used during the execution. For example, as the sequence numbers inside the nodes show the order of execution, the user can experiment with different strategies (e.g. BFS, random, fitness-guided) to observe the traversing in codes using loops or recursion.

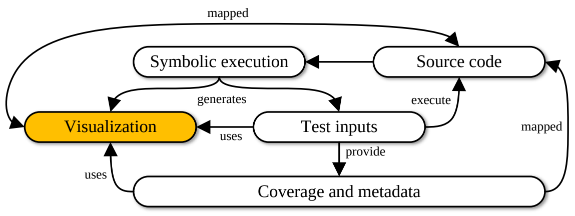
Let us see the example source code from the previous section along with the manually drawn symbolic execution tree. Microsoft Pex was run on the code, and SEVIZ monitored the execution. Fig. 4a shows the opened symbolic execution tree. SEVIZ shows how Pex explored the code. The shape of the nodes illustrate that the constraint solver was called three times (node 0, 1 and 3). Pex selected three test inputs (node 2, 4 and 5). The edge from node 1 to 3 is colored, because that edge corresponds to calling the `Bar` method, and we specified method `Foo` as the unit under test. By selecting the nodes, all the path conditions can be examined in the details panel.

Fig. 4b presents a real-world example from a method that manipulates strings. The node with purple selection has its details opened, while the orange nodes show the path, which should be traversed to get there.

B. *Engineering Usage*

Our tool supports test engineers, who are familiar with test input generation and use it on complex software. Their workflow (generate tests for code, examine generated test inputs and coverage, improve generation) can be supported by visualization as shown in Fig. 4.

Fig. 4. Symbolic execution workflow with visualization



In general, the visualization of the symbolic execution with our tool can help identifying problems, which prevent achieving higher code coverage. The following list summarizes the most important characteristics to analyze in common situations.

- *Shape of the tree:* It shows the effectiveness of the traversal strategy. Too deep branches could mean that the search strategy was not efficient or a loop or recursive structure was not properly handled.
- *Generated tests:* If there are many leaf nodes with no corresponding generated tests, those can be unnecessary runs. Either they should be cut off earlier or guidance should be provided to help selecting test inputs.
- *Path constraints:* Complex constraints blocking the progress of symbolic execution can be identified by analyzing the path conditions.
- *Unit borders:* Stepping through borders means reaching code regions, which are not in scope of the testing. Indicating these exit points supports finding correct position for isolation.

Let us revisit the motivational GitSharp example from Section II to show how visualization can be used to increase coverage. With SEVIZ we can more quickly identify the issue in the generation and avoid the detailed analysis of the reports and logs using the following steps.

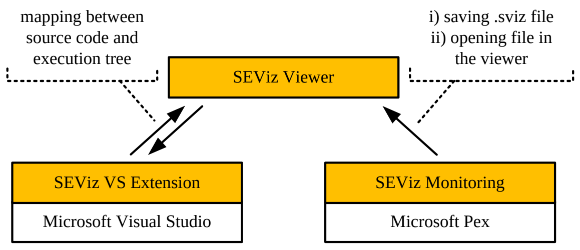
1) *Execution identification (Fig. 6a):* By just looking at the tree at a high level, we can identify that test inputs were only selected in the runs at the right side of the tree, and there is a large, deep portion of the tree where Pex spent considerable time, but was not able to select any tests.
2) *Loop identification (Fig. 6b):* In the next step, we took a look at the source code mappings of the nodes in this deep branch. We discovered, that the nodes are mapped to a loop, which causes the tree to be deep. Now we should investigate why the execution did not quit from the loop.
3) *Path condition identification (Fig. 6c):* The answer lies in the path conditions of the nodes. It can be seen, that Pex could not guess the correct condition to exit the loop, it tried to add the same characters until reaching its execution boundary.

By going through these three steps, the problem (format of the revision string) can be discovered just by working with the data presented by SEVIZ, and there is no need for further analysis or debugging.

V. IMPLEMENTATION

Although currently the tool works with data from Pex, our intention was to design a tool, which could be used with a wide range of symbolic execution-based tools. Thus, we separated the architecture into three loosely-coupled components. Each of them has well-defined tasks to support maintainability. Fig. 6 shows an overview of the structure of the tool.

Fig. 6. Architecture of the SEVIZ tool



A. SEVIZ *Monitoring*

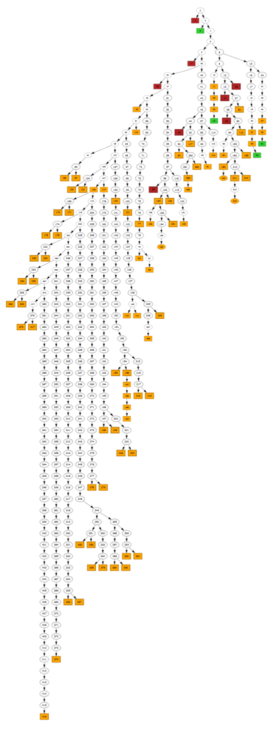
The component collects the information from the executions of Pex through the provided Pex extension API. Several extension points are present in Pex, thus we had to decide, that which one affects the executions the least. We gathered ideas for the implementation from existing, open-source Pex extensions (e.g. [13]–[15]).

Pex uses search frontiers during the symbolic execution to choose which code location (execution node) is going to be explored next. The tool allows defining search frontiers in order to alter the strategy of the code exploration. A monitoring search frontier, based on the default one, would have been a straightforward solution. However, during the implementation of our monitoring search frontier, we discovered that Pex only attaches the metadata to the nodes after the executions, which cannot be reached through this API. Thus, we searched for another solution.

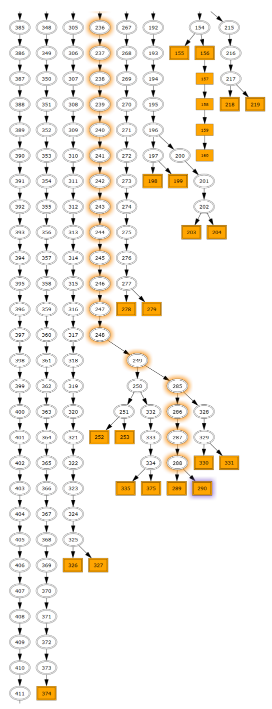
The architecture type of Pex offers extensibility in each of its layer (which is execution, exploration and path). The path layer defines callbacks before and after each execution path. Firstly, we analyzed these path-based operations with test extensions to determine if the layer fits our requirements. After an execution of a path, Pex provides the list of the executed nodes with all the required metadata attached. Nevertheless, we need other information, which can only be extracted after the whole exploration (e.g. generated tests, Z3 [16] calls).

This monitoring component stores the execution nodes in an internal storage, which plugs itself into Pex. After the

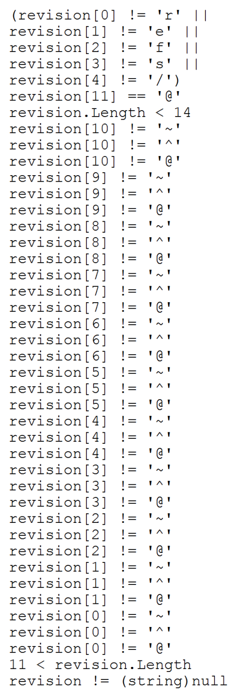
Fig. 5. Analysis steps of the GitSharp example in SEVIZ



(a) Overview of the tree



(b) Loop identification



```
(revision[0] != 'r' ||
revision[1] != 'e' ||
revision[2] != 'f' ||
revision[3] != 's' ||
revision[4] != '/')
revision[11] == '@'
revision.Length < 14
revision[10] != '~'
revision[10] != '^'
revision[10] != '@'
revision[9] != '~'
revision[9] != '^'
revision[9] != '@'
revision[8] != '~'
revision[8] != '^'
revision[8] != '@'
revision[7] != '~'
revision[7] != '^'
revision[7] != '@'
revision[6] != '~'
revision[6] != '^'
revision[6] != '@'
revision[5] != '~'
revision[5] != '^'
revision[5] != '@'
revision[4] != '~'
revision[4] != '^'
revision[4] != '@'
revision[3] != '~'
revision[3] != '^'
revision[3] != '@'
revision[2] != '~'
revision[2] != '^'
revision[2] != '@'
revision[1] != '~'
revision[1] != '^'
revision[1] != '@'
revision[0] != '~'
revision[0] != '^'
revision[0] != '@'
11 < revision.Length
revision != (string)null
```

(c) Path condition of a selected node

exploration ends, SEVIZ flushes the collected nodes into a GraphViz [17] file and all of its related metadata into data files. Then, SEVIZ calls GraphViz to compile its source to create a layout for the graph. Finally, the generated files are zipped into a SEVIZ (.sviz) file, which can be opened with the SEVIZ Viewer.

SEVIZ Monitoring is compatible with the public version of Microsoft Pex (v0.94.51023.0), and also successfully collaborates with the recently released Smart Unit Tests integrated in Visual Studio 2015 Preview.

Our preliminary measurement showed that the overhead of the monitoring depends on the structure of the code under test instead of its length. Significant overhead was mainly observed only in cases of uncontrolled loops.

B. SEVIZ *Viewer*

The main component of the architecture is based on GraphViz files. We used an open-source project (DotViewer [18]) to visualize the compiled graph data. The Viewer opens SEVIZ (.seviz) files, extracts its content, and builds up the graph. The DotViewer WPF control enables to zoom in and out, and also allows to select one or multiple nodes. The main features besides the visualization are the following.

- *Zoom and scroll:* Users are able to zoom into large graphs to search for details, and it is also allowed to scroll in two dimensions.
- *Node selection:* Users can select one or multiple nodes for analysis. These are highlighted in the visualized graph.
- *Node details:* Detailed metadata of the nodes are presented in two forms. 1) When the cursor hovers a node, then a tooltip appears with the incremental path condition, the method of the node and the code location mapping (if it exists). 2) When a node is selected, all of its metadata is presented in the node details section of the UI, which can be opened in a new window too. This enables users to compare the data of different nodes.
- *Run selection:* SEVIZ enables users to select a run from a list and the tool highlights the corresponding nodes in the visualized graph.

These features almost fulfills the requirements, which we established in Section II, although the source code mapping is missing, which we describe below in detail.

C. SEVIZ *Visual Studio Extension*

The third component, an extension for Visual Studio, is responsible for ensuring the two-way mapping between the source code and the visualized nodes in SEVIZ Viewer.

The component uses pipes to implement inter-process communication with the Viewer. We created a pipe server in the Viewer and also in the Visual Studio extension for ensuring two-way communication. Thus, when a user selects a source code line the corresponding node is selected in the symbolic execution graph. Furthermore in the other direction, when a user selects a node in the graph, the corresponding source code line is selected in a predefined Visual Studio instance.

The difficulty of the implementation was that Pex uses IL, the intermediate language of .NET, for symbolic execution.

To implement this mapping, we relied on the internal services of Pex, which are made available through its API. First, we get the instrumentation info about the method of the currently analyzed node. Then, we extract the offset of the node in this method, which results in a sequence point. Finally, this can be used in the translation to the source code. However, this multi level mapping does have some potential inaccuracy, which can be optimized in future works.

VI. Related Work

Currently, test generation techniques, like symbolic execution, are getting into a promising research state, which leads to the increase of industrial application willingness. Thus, this topic is also a current aspect of the research around symbolic execution. Tillmann *et al.*, the developers of Microsoft Pex, reported on their experiences [10] with the technology transfer from research prototype to a mature test generation tool, which can be used in numerous real-world scenarios. Challenges occurring around the application of symbolic execution in large-scale software motivate researches like parallelization [19], the idea of Micro Execution [20], or the techniques rounded up by Cadar *et al.* in [3]. Micro execution alleviates the problem of testing in isolation, which is a common problem in real-world test generation. Parallelization of symbolic execution raises new problems as well. A possible solution was introduced in [21].

Several papers utilize the visualization of symbolic execution in different contexts. Hahnle *et al.* implemented a tool, which can be used to debug applications visually with the usage of a symbolic execution tree [22]. They used a different representation in the visualization than ours, specialized for debugging and not for test generation purposes. Despite its benefits, visualization raises numerous problems, especially in cases, where symbolic execution has large or even infinite space to discover (which can not be visualized). For example, representing loops and recursions in symbolic execution trees are included in the topics of research around the visualization. Hentschel *et al.* introduced a method, which can compose the symbolic execution tree finitely in case of loops or recursions by using method summaries [23]. This approach can be a powerful extension of our visualization tool, since loops and recursions are part of our further and planned improvements.

VII. Conclusion and Future Work

In this paper we introduced SEViz, a tool for visualizing symbolic execution. Our tool enhances symbolic execution-based test generation in large-scale software by providing in an intuitive way the necessary information to increase the achieved coverage of the generated test inputs. The tool currently works with Microsoft Pex, however the architecture provides the ability to extend the list of compatible tools.

We presented the functions and capabilities of SEViz through a real-life example by testing a small part of the open-sourced GitSharp project. In this example, we demonstrated how to use the tool for achieving higher code coverage without effort-intensive and time-consuming report analysis.

Currently, our tool is in a prototype phase, therefore we have several plans on future work: i) mapping the symbolic execution tree into control flow graphs of each executed method, ii) extending the set of compatible symbolic execution-based tools and development environments, iii) controlled empirical evaluation of the tool to confirm its effectiveness and usability.

Acknowledgment

We would like to thank the comments and suggestions of the anonymous reviewers.

This work was partially supported by the ARTEMIS JU and the Hungarian National Research, Development and Innovation Fund in the frame of the R5-COP project.

References

[1] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, “An Orchestrated Survey of Methodologies for Automated Software Test Case Generation,” *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, 2013.

[2] J. C. King, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[3] C. Cadar and K. Sen, “Symbolic Execution for Software Testing: Three Decades Later,” *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[4] T. Chen, X. song Zhang, S. ze Guo, H. yuan Li, and Y. Wu, “State of the art: Dynamic symbolic execution for automated test generation,” *Future Gener. Comp. Sy.*, vol. 29, no. 7, pp. 1758 – 1773, 2013.

[5] C. Barrett, A. Stump, and C. Tinelli, “The Satisfiability Modulo Theories Library (SMT-LIB),” http://www.smt-lib.org, 2010.

[6] N. Tillmann and J. de Halleux, “Pex–White Box Test Generation for .NET,” in *Tests and Proofs*, ser. LNCS, B. Beckert and R. Hähnle, Eds. Springer, 2008, vol. 4966, pp. 134–153.

[7] X. Qu and B. Robinson, “A Case Study of Concolic Testing Tools and their Limitations,” in *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, Sept 2011, pp. 117–126.

[8] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, “Does automated white-box test generation really help software testers?” in *Proc. of the 2013 Int. Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 291–301.

[9] P. Braione, G. Denaro, A. Mattavelli, M. Vivanti, and A. Muhammad, “Software testing with code-based test generators: data and lessons learned from a case study with an industrial software component,” *Software Quality Journal*, vol. 22, no. 2, pp. 1–23, 2013.

[10] N. Tillmann, J. de Halleux, and T. Xie, “Transferring an Automated Test Generation Tool to Practice: From Pex to Fakes and Code Digger,” in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE ’14. New York, NY, USA: ACM, 2014, pp. 385–396.

[11] eqqon GmbH, “GitSharp,” http://www.eqqon.com/index.php/GitSharp, 2013.

[12] D. L. 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.

[13] C. Csallner, N. Tillmann, and Y. Smaragdakis, “DySy: dynamic symbolic execution for invariant inference,” in *Proc. of the 13th international conference on Software engineering - ICSE ’08*, 2008, pp. 281–290.

[14] X. Xiao, T. Xie, N. Tillmann, and J. D. Halleux, “Covana : Precise Identification of Problems in Pex,” in *Software Engineering (ICSE), 2011 33rd International Conference on*, 2011, pp. 1004–1006.

[15] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, “Fitness-guided path exploration in dynamic symbolic execution,” in *Dependable Systems Networks, 2009. DSN ’09. IEEE/IFIP Int. Conf. on*, 2009, pp. 359–368.

[16] L. de Moura and N. Bjorner, “Z3: An Efficient SMT Solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS. Springer, 2008, vol. 4963, pp. 337–340.

[17] E. R. Gansner and S. C. North, “An open graph visualization system and its applications to software engineering,” *SOFTWARE - PRACTICE AND EXPERIENCE*, vol. 30, no. 11, pp. 1203–1233, 2000.

[18] C. Rodemeyer, “Dot2WPF - a WPF control for viewing Dot graphs,” 2007. [Online]. Available: http://www.codeproject.com/Articles/18870/Dot-WPF-a-WPF-control-for-viewing-Dot-graphs

[19] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, “Parallel symbolic execution for automated real-world software testing,” in *Proc. of the sixth conference on Computer systems (EuroSys’11)*, 2011, pp. 183–198.

[20] P. Godefroid, “Micro Execution,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 539–549.

[21] D. Vanoverberghe and F. Piessens, “Theoretical Aspects of Compositional Symbolic Execution,” in *Fundamental Approaches to Software Engineering*, ser. LNCS, D. Giannakopoulou and F. Orejas, Eds. Springer Berlin Heidelberg, 2011, vol. 6603, pp. 247–261.

[22] 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*, ser. ASE ’10. ACM, 2010, pp. 143–146.

[23] M. Hentschel, R. Hähnle, and R. Bubel, “Visualizing Unbounded Symbolic Execution,” in *Tests and Proofs*, ser. LNCS, M. Seidl and N. Tillmann, Eds. Springer, 2014, vol. 8570, pp. 82–98.