Visualizing Path Exploration to Assist Problem Diagnosis for Structural Test Generation

Jiayi Cao[1], Angello Astorga[1], Siwakorn Srisakaokul[1], Zhengkai Wu[1], Xueqing Liu[1], Xusheng Xiao[2], Tao Xie[1]
[1]University of Illinois at Urbana-Champaign, [2]Case Western Reserve University
Email: {jcao7,aastorg2,srisaka2,zw3,xliu93,taoxie}@illinois.edu, xusheng.xiao@case.edu

***Abstract*—Dynamic Symbolic Execution (DSE) is among the most effective techniques for structural test generation, i.e., test generation to achieve high structural coverage. Despite its recent success, DSE still suffers from various problems such as the boundary problem when applied on various programs in practice. To assist problem diagnosis for structural test generation, in this paper, 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.**

I. Introduction

Dynamic Symbolic Execution (DSE) [1]–[3] is among the most effective techniques for structural test generation [4]. DSE collects the constraints on inputs from executed branches to form path conditions and flips constraints in the path conditions to obtain new path conditions for exploring new paths and achieving high structural coverage. However, users of DSE-based tools such as Pex [5]–[7] often experience various categories of problems [8]–[11] while applying the tools on various programs in practice. One major category of problems is the 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 due to insufficiency of its default exploration-resource allocation (such as the maximum number of explored paths allocated to path exploration). Such problem often occurs for a program under test containing loops [12] or complex string operations [13].

When such problem arises, the DSE-based tools present little information about the problem root cause, leaving the users in the dark. Furthermore, there is little visual (i.e., easy to digest) guidance readily available to solve the problem 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 problem root cause is prohibitive.

To address such issue, in this paper, 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 particular, our approach provides visualization to summarize the path-exploration results by collapsing redundant exploration results through a Variant Control Flow Graph (VCFG) (a CFG with its nodes being reduced to only those corresponding to branch statements) 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. We implement our approach as an extension to IntelliTest (derived from Pex [5]–[7]), an industrial test generator available in Visual Studio 2015/2017, and a significant improvement over an existing state-of-the-art visualization approach, SEViz [14].

978-1-5386-4235-1/18/$31.00 ©2018 IEEE

II. PexViz Approach

Our PexViz approach consists of three components: the Variant Control Flow Graph (VCFG) generator, the exploration-data augmentor, and the graph visualizer. The VCFG generator reads the program source code and transforms it into a VCFG representation, with an example shown in Figure 2. 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 starting 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. Example information includes incremental path condition, being the predicate gathered from the branch statement corresponding to the current VCFG node, and flip count, being the count of flipping the constraint gathered from the incremental path condition of the current VCFG node. Finally, the graph visualizer reads the output VCFG and generates an interactive visualization front-end to present information to the users. We next illustrate the details of the graph visualizer, with a modified `BubbleSort` example.

A. Graph Visualizer

The graph visualizer includes the visualization front-end to display the VCFG graph and information on it, with an example shown in Figure 2. 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

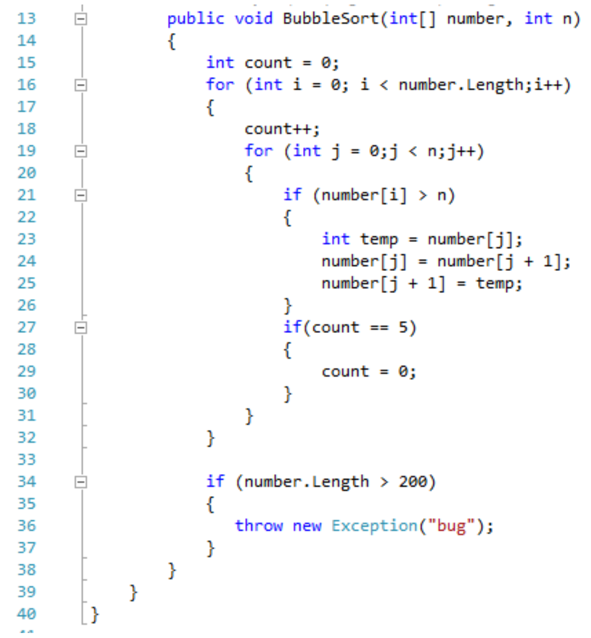
Boolean predicate within the branch statement as the label for the VCFG node so that the users can quickly identify which line of code the branch statement belongs to. In case there are the same or similar branch conditions from the source code for multiple VCFG nodes, the users can click on each VCFG 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 another. 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 exploration-data augmentor, the graph visualizer renders the information into filled colors, text labels, and textual data. In particular, the following information is visualized:

- Shape. A rectangle represents a VCFG node for a branch statement in the source code while a circle represents a utility VCFG node, such as an entry point.
- Filled color. (1) White represents that the incremental path condition in the VCFG node does not contain symbolic variables. White indicates lower inspection priority. (2) Green represents that the incremental path condition contains 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. (3) Orange represents an un-flipped constraint from an incremental path condition that contains symbolic variables. Orange is a warning color to indicate a threatening factor. (4) Red represents an unreached branch statement during the DSE exploration. Red indicates a serious situation deserving attention. (5) Blue represents a utility VCFG node, such as an entry node, which is a node that does not come from the source code.

B. *Example*

Figure 1 shows a code snippet adapted from a bubble sort method in the open source DSA project (https://archive.codeplex.com/?p=dsa). We run IntelliTest with default settings on the code snippet and obtain 11/14 block coverage. The IntelliTest console result indicates that 122 paths have been explored until IntelliTest stops because of reaching the timeout boundary, and IntelliTest generates 6 inputs that cover different blocks, along with 2 inputs that trigger exceptions.

The tool users can investigate the corresponding PexViz graph as shown in Figure 2. There are 6 VCFG nodes in the PexViz graph, which has 97.8% fewer nodes than the 276 nodes from the SEViz [14] graph (not shown here due to space limit). The users can start examining the PexViz graph from the blue entry VCFG node. The users can directly observe the clear correspondence between VCFG nodes and branch statements through the VCFG node labels. Thus, such mechanism saves the users navigation time in contrast to clicking through each of the 276 nodes in the SEViz graph.



```
        public void BubbleSort(int[] number, int n)
        {
            int count = 0;
            for (int i = 0; i < number.Length;i++)
            {
                count++;
                for (int j = 0;j < n;j++)
                {
                    if (number[i] > n)
                    {
                        int temp = number[j];
                        number[j] = number[j + 1];
                        number[j + 1] = temp;
                    }
                    if(count == 5)
                    {
                        count = 0;
                    }
                }
            }

            if (number.Length > 200)
            {
               throw new Exception("bug");
            }
        }
    }
}
```

Fig. 1: A code snippet of modified `BubbleSort` where a boundary problem is faced

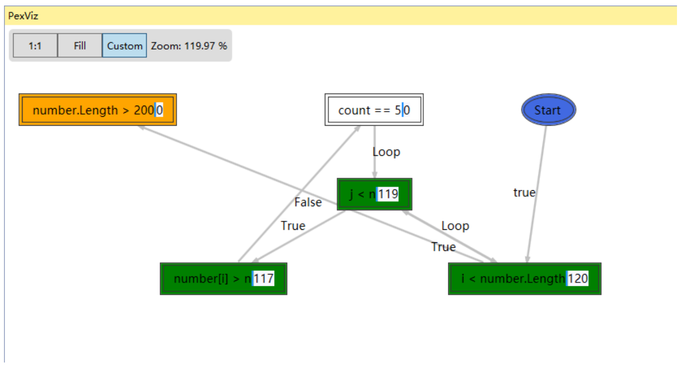


Fig. 2: PexViz visualization on on running IntelliTest against the modified `BubbleSort`

The orange VCFG node showing 0 flip count immediately draws the users' 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 constraint is not flipped, the users can examine the three neighboring green VCFG nodes. All three green VCFG nodes have flip count of around 120. The information on the generated test inputs that the users can observe after clicking on the three green VCFG nodes indicates that the array with length larger than 200 is not created. IntelliTest stops before it is able to generate an array with length 200; therefore, increasing the bound of the maximum number of explored paths can be a solution to the problem. After the users increase the bound, IntelliTest manages to reach 14/14 (100%) block coverage.

Acknowledgment. This work was supported in part by National Science Foundation under grants no. CNS-1513939 and CNS-1564274.

References

[1] 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.

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

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

[4] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Future of developer testing: Building quality in code," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER 2010)*, 2010, pp. 415–420.

[5] 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.

[6] 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.

[7] 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 ACM/IEEE international conference on Automated Software Engineering (ASE 2014)*, 2014, pp. 385–396.

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

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

[10] 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.

[11] 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.

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

[13] N. Li, T. Xie, N. Tillmann, J. d. Halleux, and W. Schulte, "Reggae: Automated test generation for programs using complex regular expressions," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, 2009, pp. 515–519.

[14] 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.