Towards Supporting Dynamic Symbolic Execution via Multi-Domain Metrics

Dávid Honfi, Zoltán Micskei
Department of Measurement and Information Systems,
Budapest University of Technology and Economics,
Budapest, Hungary
Email: {honfi,micskeiz}@mit.bme.hu

***Abstract*—A popular code-based test generation technique is dynamic symbolic execution (DSE), which combines concrete executions with symbolic ones. The current maturity of DSE led to industrial usages. However, the complexity of software used in industrial practice poses several challenges for DSE. The issues caused by these are often hard to identify as they are mostly indicated by only the lack of coverage. In this paper, we gather and present metrics that can be used on top of our already elaborated approach that visualizes symbolic executions trees.**

I. Introduction

Today, testing is an inevitable task of software development. Academic research also tackles the topic from various aspects. As test development is a time-consuming task, it is beneficial to introduce automation. Several approaches have been proposed addressing this challenge, including ones automatically generating tests from source code (white-box test generation). One of these techniques is *symbolic execution* (SE).

Symbolic execution is a widely used technique originating from the '80s. It begins the execution on a given entry point in the program. SE replaces the concrete variables with symbolic ones and uses them to form path constraints on each known execution path (path condition). Then, these constraints are transformed to SMT problems, which can be solved using special-purpose tools. The concrete values satisfying the path constraints steer the program execution exactly to the path corresponding to the given constraint.

Dynamic symbolic execution (DSE) enhances the original technique by mixing it with concrete executions. DSE starts a concrete execution from an arbitrary entry point with the simplest concrete input values as possible, while symbolic execution is performed in parallel. During the concrete execution, SE collects the constraints on the given path. When a path in an execution has finished, the DSE transforms (e.g., negates) the collected constraint system and then attempts to obtain a solution by defining it as a Satisfiability Modulo Theories (SMT) problem. If a solution is available, then a new concrete execution is feasible with the new inputs. This process continues until either no more feasible paths are available or an execution boundary (e.g., time limit) is reached.

Both SE and DSE face several challenges in numerous cases. This includes, for instance, the following [1].

- *Constraint solver issues (CSI):* There are typical issues for constraint solvers such as formulas containing floating point arithmetics due to the high precision representation, or large path constraints with diverse types of variables.
- *State space explosion (SSE):* When dynamic symbolic execution is unbounded, the algorithm may explore program states that are out-of-scope or cause fruitless executions (e.g., redundant paths). However, when a boundary is set, it should be defined in a way that it does not hinder exploring important states in the program.
- *Object creation (OC):* A common issue in complex programs is that the objects passed as parameters must be assembled through sequences of method calls or using special factory methods. These are usually hard to guess automatically for a symbolic execution engine, which prevents the program exploration. In these cases, manual intervention or specialized algorithms shall be used.
- *Environment interactions (EI):* Interactions with the environment of the unit under analysis are commonly found in complex software. These interactions are mostly calls to databases, to network or to the file system. Furthermore, there could be invocations to underlying frameworks. Omitting the handling of these calls may lead to undesired behaviors (e.g., writing to file system or database).

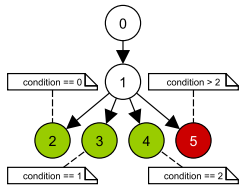
These challenges often cause issues that result in not enough tests being generated. However, identifying and localizing the root causes of the occurring issues may require an excessive amount of effort. The effort spent on the identification may reduce the advantages gained from automation.

To alleviate the identification and localization process, we have already presented an approach that visualizes the symbolic execution [2]. This technique uses an internal representation of the execution called symbolic execution tree and visualizes it with a predefined semantics. The use case of the visualization is twofold: it gives an overall overview of the execution, and gives internal details at each symbolic state (e.g., path condition). In our visualization, we enriched the symbolic execution tree nodes with various metadata: sequence number, location, corresponding runs, path condition, constraint solver calls, generated tests (for leaf nodes). An example code and the corresponding symbolic execution tree is shown in Figure 1.

The attached metadata serves as the basis for identifying

```
public int SwitchBranching(int condition) {
    var divisor = 0;
    switch(condition) {
        case 0: return 0;
        case 1: return -1;
        case 2: return -2;
        default: return (condition / divider);
    };
}
```

(a) The source of SwitchBranching.



(b) The symbolic execution tree.

Fig. 1. A simple method and the corresponding symbolic execution tree extracted from SEViz. The green leaves indicate passing generated tests, while the red leaf shows a test generated for the path, which raises an exception.

the issues of the test generation process. However, in a wider focus area, there is a large number of other metrics that can be attached to a symbolic execution tree. These metrics can be obtained from related domains (e.g., source codes, graphs). With the additional metrics attached, a symbolic execution tree may be more capable of indicating root causes of challenging issues. Moreover, the data obtained for the metrics can be used for recommendations and predictions.

The goal of this paper is to gather from literature, present and describe valuable metrics attached to symbolic execution trees. The metrics we set out to present are obtained from various sources and domains to improve diversity. Furthermore, another goal in the paper is to present the applicability of the metrics on an artificial example.

The rest of the paper is organized as follows. In Section II we present the collected metrics in detail including their original domains. Section III shows examples on how to support test generation and identify issues using the gathered metrics. Section IV discusses the related work, while Section V concludes our contributions.

II. Metric Selection

Source. To gain an overview of what metrics can be attached to a symbolic execution tree, we defined four categories from related domains of dynamic symbolic execution-based test generation. For each of the categories, we searched for survey papers that collect the most important and widely-used metrics in their domains. The domains we selected are source code, dynamic symbolic execution, tests, and graphs.

Context. In this paper, we select and detail four most relevant metrics for each category. Also, we provide indications on which metric could be influenced by the issues stated before and vice versa (denoted with the abbreviation of the issue). Albeit the selected metrics could be used as standalone indicators for issue prediction, our paper only focuses on using them for extending symbolic execution trees to perform post-analysis. We defined three *locations*, where a metric can be attached to a symbolic execution tree. We indicate these next to the name of the metric.

- Nodes (N): A node in a symbolic execution tree represents a program state. Each node is mapped to a given location of the program (as a basic block).
- Paths (P): A path is a sequence of nodes in the SE tree that represents a corresponding execution.
- Exploration (E): The exploration refers to the set of all the paths that have been executed. Basically, this is the whole symbolic execution tree.

Selection. We defined the selection criterion of the metrics based on two dimensions: 1) challenges occurring in dynamic symbolic execution and 2) locations where a metric can be placed in a symbolic execution tree. Each of the examined metrics was labeled with values from these dimensions. The defined coverage criterion requires to cover all the combinations of the *locations-challenges* dimensions with at least one metric. The labeling was based on our experiences and intuitions regarding with symbolic execution trees [2].

A. Static code-based metrics (SC)

The domain of source code can be viewed from various aspects, e.g., abstract representations like abstract syntax trees of control-flow graphs. We obtained the source code metrics from two papers. One of them is a study conducted to extract characteristics of 147 open-source Java projects [3], while the other compares programmer opinions to complexity [4].

1) Lines of Code [E][SSE]: The LoC is a textual metric of the source code measuring the number of lines. A very large program with many modules interacting with each other could lead to huge path constraints causing constraint solvers to fail.

2) Cyclomatic complexity [E][CSI, SSE]: CC is a metric that indicates the number of linearly independent paths in the source code. The number of these paths are derived from the control-flow graph (CFG) of the program. Large CC could indicate the presence of loops that would yield large path conditions and search space. Both hinder constraint solvers.

3) Halstead's difficulty [E][CSI, SSE]: The metric is sometimes called the error-proneness, which is regarding with the number of unique operators and operands in the code. Both the operators and operands are usually expanding the search space that the DSE interpreter must explore.

4) Number of method calls [N, P, E][SSE, EI]: This metric is an indicator how many calls are performed to any other methods (e.g., to other classes, libraries, environment). The larger the number of external method calls, the higher the probability of environment accessing issues for DSE. Obviously, if there are other, additional methods to explore, the search space also grows.

B. Dynamic symbolic execution metrics (SE)

In dynamic symbolic execution, one of the key concepts is the path condition that is solved by constraint solvers on each path [3]. Also, an important feature of the explored paths is the variety of instructions found along each path [5].

1) Path condition length [N,P][CSI, OC]: Represents the length of the constraint system collected on an execution path (sum of variables and operators used). Note that this metric does not care about the repeated use of the same variable, constant or operator.

2) Number of variables in path condition [N,P][CSI]: The metric measures the number of distinct variables in the path condition. This represents how dependent is the outcome of the execution path on the symbolic variables.

3) Number of constants in path condition [N,P][CSI, OC]: The metric measures the number of distinct constants in the path condition including all data types that support constants. This denotes how restrictive is the program code in the given path on the variables.

4) Path description vector [P][SSE, EI]: An executed path can be represented as a sequence of interpreted basic blocks. Each basic block contains a given number of low-level instructions, which can be represented using an occurrence vector with a fixed length (based on the number of possible instruction types on the given platform). Then, for each path, a feature matrix can be assembled using the occurrence vectors obtained along the path. This matrix represents the covered program features on the given path [5].

C. Generated test metrics (GT)

At each leaf node of the symbolic execution tree, a test case can be derived that executes the corresponding path, which ends in that node. When a test case is produced, the characteristics of the test (regarding with the inputs, actions and expectations) may give overview of the given path about what is performed along [6], [7].

1) Number of assertions [P][SSE, OC, EI]: This metric tells how many assertions are found in the given generated tests. If there are too many, then the test may be too specific, which could cause false positive outcomes and could lead to Assertion Roulette [8]. On the contrary, if there are few assertions, then it could mean that the test case is too permissive and may omit to check important behaviors. This issue is usually caused by a problem in the DSE engine about what observed behaviors to check.

2) Number of different types of assertions [P][SSE, OC, EI]: It measures the variety of assertions in a generated test. It may indicate that the single test case is checking multiple behaviors. However, a high number for this metric may indicate that the test is too specific. This is usually caused by an issue on observing the behavior of the program.

3) Lines of test code [P][OC]: The length of the test code is usually a good indicator of its complexity. Too long tests may hinder easy understanding, or it could contain unwanted setups or assertions. Long tests may also indicate that the dynamic symbolic execution engine was only able to setup the test environment in an unusual or unnecessary way.

4) Number of constants in test code [P][CSI, SSE]: The metric measures the number of constant values that the dynamic symbolic execution engine was able to generate into the code. If there is a value, it means that the algorithm was able to discover and parse it from a given basic code block. A large number of constants in the code usually yields wrongly handled unbounded loops in the program.

D. Generic graph metrics (GG)

It is typical to apply general metrics for a wide variety of graphs across several domains. This is the case for software engineering as well [9]. The most common abstract representation of a program is its control-flow graph that is by definition contains the possible execution paths between the basic code blocks in the program. However, to our best knowledge, there is no study, which maps these general graph metrics to symbolic execution trees.

1) Average branching factor [E][CSI, SSE, OC]: Let graph G be the symbolic execution tree with a set of vertices $V(G)$. We define the branching factor $d_+(v)$ for each node $v \in V(G)$ as its number of outgoing edges. The overall metric for the whole tree is an average calculated as $\frac{1}{|V(G)|} * \sum_{v \in V(G)} d_+(v)$. If a symbolic execution tree has a low branching factor, it could reveal that the constraint solver faced issues during the solution of complicated path conditions. High branching factor could indicate the presence of unbounded loops.

2) Height of tree [E][SSE]: The height of a tree is given by the length of the longest path selected out of all possible paths from the root to a leaf. A suspiciously deep symbolic execution tree may indicate that the dynamic symbolic execution engine was not able to appropriately handle bounds of execution.

3) Number of leaves [E][SSE]: The number of leaves in a tree provides a way to determine its width. As a leaf node represents the end of an execution path, we use this metric to decide how many tests could have been generated. The metric should be used in strong cooperation with others (e.g., test outcomes, branching factor).

4) Diameter of the tree [E][SSE]: To determine the diameter of a symbolic execution tree, we temporarily remove the directions of edges and calculate the longest path available among all of the node pairs. From the longest paths between all of the node pairs, we select longest one to indicate the diameter of the whole tree. If the diameter of a symbolic execution tree is unusually large, it may yield that search space is too huge for the engine and must be handled properly.

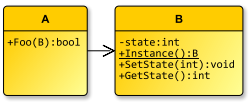
In Table I we summarize the coverage of the metrics on both of the defined dimensions. It can be seen that although the coverage criterion has been fulfilled, yet there are some fields, which have a smaller amount of associated metrics (e.g., object creation issue identification on node level).

TABLE I
SUMMARIZING TABLE OF METRIC COVERAGE OF THE TWO DIMENSIONS DEFINED (LOCATION-CHALLENGE).

	CSI	**SSE**	**OC**	**EI**
N	SE-1, SE-2, SE-3	SC-4	SE-1, SE-3	SC-4
P	SE-1, SE-2, SE-3, GT-4	SC-4, SE-4, GT-1, GT-2, GT-4	SE-1, SE-3, GT-1, GT-2, GT-3	SC-4, SE-4, GT-1, GT-2
E	SC-2, SC-3, GG-1	SC-1, SC-2, SC-3, SC-4, GG-1, GG-2, GG-3, GG-4	GG-1	SC-4

III. AN EXAMPLE USE OF THE METRICS

We present how the metrics could indicate issues through an example. In this simple program, the identification of an object



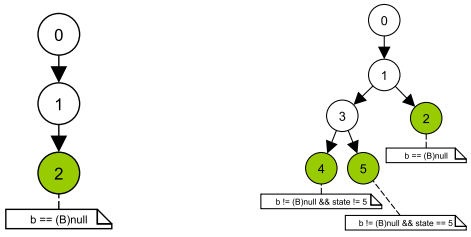

```
public bool Foo(B b) {
    if(b == null) return false;
    if(b.GetState() == 5) return true;
    return false;
}
```

(a) Class diagram of the example. (b) Example method for analysis.

Fig. 2. The example program architecture and code with classes A and B.

creation issue (OC) will be demonstrated via the attached metrics. Consider the example class layout shown in Figure 2a. Class `A` contains the current method under test `Foo` that has an argument of type `B`. Class `B` has a private default constructor and a method (`Instance`) that obtains an instance of this class. We executed dynamic symbolic execution on method `Foo` using Microsoft Pex, a state-of-the-art DSE-based test generator [10]. The outcome visualized in SEViz is shown in Figure 3a. We identified the number of constants in the path condition (II-B3), the lines of test code (II-C3) and the branching factor (II-D1) as a unique indicator set of the OC.

Figure 3b shows the symbolic execution tree after the OC issue has been resolved using a manual factory method. The branching factor of the tree previously was 1, while it increased to 1.66 with using the factory. Furthermore, the generated test code length was 5 LoC in the first case, which increased to 6.33 in the second case. The number of constants in the path condition also confirms the issue: in the first case, there is only one constant used in the only path (`null`), while in the extended case, there are other constants as well (e.g., `state == 5`). Only null constants in the path conditions throughout the tree usually indicate that there is a problem with creating objects for the given variable. In this example, this hypothesis was supported by two facts: the low branching factor of the tree and test cases with short length. The identification of the root cause is fairly simple using the path condition variables, therefore can be automated: one shall examine, which variables were only constrained to null. T



(a) SE tree with OC issue present. (b) SE tree after resolved OC issue.

Fig. 3. SE trees for the example with the OC issue present and resolved.

IV. Related work

A related technique is Covana [11] that aims to identify OC and EI issues. Covana monitors DSE and collects problem candidates that are examined using data dependence analysis. While this approach is based on runtime monitoring, our approach uses only metrics attached to previously produced symbolic execution trees. SED is a symbolic execution debugger that visualizes symbolic execution trees with metadata [12]. The main difference between their approach is that they use it for debugging, while our approach aims at identifying issues of DSE-based test generation. Baldoni *et al.* survey symbolic execution techniques along with identifying their challenges [1]. They also consider possible solutions to these problems. Our technique may be extended with advising solutions to identified issues. Eler *et al.* analyzed characteristics of Java programs influencing the performance of symbolic execution [3]. We used some of their defined metrics to attach them to the nodes of the generated symbolic execution trees (representing a program state and location).

V. Conclusions and future work

In this paper, we gathered and presented metrics from various domains to alleviate the issues of dynamic symbolic execution based on previously extracted symbolic execution trees. We selected 16 metrics from papers of 4 related domains based on a predefined coverage criterion to enhance the problem identification process. The metrics have been presented in detail along with two metadata for each of the metrics indicating that 1) for which issue they can be used and 2) where they can be attached in the symbolic execution tree. We also presented an example issue, where the metrics have perceivable changes caused by the presence of the given issue.

Our future work is twofold: 1) we plan to extend the set of collected metrics in a more systematic way, 2) we elaborate a technique providing automated identifications of DSE problems using the metrics attached to symbolic execution trees.

References

[1] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *CoRR*, vol. abs/1610.00502, 2016. [Online]. Available: http://arxiv.org/abs/1610.00502

[2] D. Honfi, A. Voros, and Z. Micskei, "SEViz: A tool for visualizing symbolic execution," in *ICST*, April 2015, pp. 1–8.

[3] M. M. Eler, A. T. Endo, and V. H. Durelli, "An empirical study to quantify the characteristics of Java programs that may influence symbolic execution from a unit testing perspective," *Journal of Systems and Software*, vol. 121, pp. 281 – 297, 2016.

[4] B. Katzmarski and R. Koschke, "Program complexity metrics and programmer opinions," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, 2012, pp. 17–26.

[5] R. P. L. Buse and W. Weimer, "The road not taken: Estimating path execution frequency statically," in *31st IEEE International Conference on Software Engineering*, 2009, pp. 144–154.

[6] V. Garousi and M. Felderer, "Developing, verifying, and maintaining high-quality automated test scripts," *IEEE Software*, vol. 33, no. 3, pp. 68–75, 2016.

[7] D. Bowes, T. Hall, J. Petrić, T. Shippey, and B. Turhan, "How good are my tests?" in *Proceedings of the 8th Workshop on Emerging Trends in Software Metrics*. IEEE Press, 2017, pp. 9–14.

[8] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006.

[9] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *ICSE*. IEEE, 2012, pp. 419–429.

[10] N. Tillmann and J. de Halleux, "Pex–white box test generation for .net," ser. TAP: Second International Conference, 2008, pp. 134–153.

[11] X. Xiao, T. Xie, N. Tillmann, and J. De Halleux, "Precise identification of problems for structural test generation," in *ICSE*. IEEE, 2011, pp. 611–620.

[12] R. Hähnle, M. Baum, R. Bubel, and M. Rothe, "A visual interactive debugger based on symbolic execution," in *ASE*, 2010, pp. 143–146.