Proceedings of the 10th International Conference on Applied Informatics
Eger, Hungary, January 30–February 1, 2017. pp. 245–253
doi: 10.14794/ICAI.10.2017.245

Applying Heuristics to Improve our Java Symbolic Execution Engine*

Edit Pengő

Department of Software Engineering, University of Szeged
pengoe@inf.u-szeged.hu

Abstract

Nowdays many static analyser tools are available to aid software development and testing by detecting erroneous code segments and runtime issues. Symbolic execution is a type of static analysis that can be used for exhaustive runtime error search without the actual execution of the analysed program.

We developed a symbolic execution engine, called RTEHunter, for detecting runtime failures in Java programs. As in the case of other symbolic engines, overcoming the problems connected to path explosion and constraint solving is an important challange for RTEHunter as well. Our goal is to improve the results of RTEHunter by finding a heuristical constraint solving approach. We describe the solution and present the results produced on more than 200 various sized Java systems.

Keywords: Source code analysis, Symbolic execution, Java

MSC: 68N01

1. Introduction

Testing is a critical part of software development for identifying bugs and runtime issues. However when deadlines come closer or new requirments arise, testing is often neglected. It is because testing – especially sophisticated, high coverity manual testing – requires a significant amount of resources which is not always available during the pressure of production. A solution to this problem is to introduce more automated methods that support bug detection.

Static analyser tools provide automated help in finding issues by analysing the source code itself. Modern IDEs contain such tools to aid quality software development and also many open source and commercial static analyser programs are available. The time and computational effort required for the analysis depend on

*This research was supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled "Internet of Living Things".

the complexity of the aimed issues. IDE integtared and open-source tools usually perform quick analysis and detect problems like dead code, whilst for serious problems a more time-consuming deep static analysis is needed. The classical symbolic execution performs this kind of complex analysis by simulating the execution of the program with the use of static analysis information only. This way it is possible to find more severe runtime issues that might have remained undiscovered even after manual testing.

In this work, we focus on the improvement of the symbolic execution engine developed by our team. Our goal was to reduce its resource requirements without notably altering its results. We examined that constraint solving is a considerable challange, therefore we applied our heuristical approaches on this field. The solution was tested on more than two hundred open source Java systems and it was proven that in practice it worked well.

The rest of the paper is organized as follows. Related work is discussed in Section 2. Section 3 introduces the technical background of classical symbolic execution and our static analyzer. The improvement and the results are presented in Section 4. Finally, we conclude the study in Section 5.

2. Related Work

The fundamentals of symbolic execution was introduced by King [9] in 1976. Another important early work is the survey of Coward [4]. He revealed the major limitations and challanges of symbolic engines that still need to be addressed nowdays too. Due to the growth in computational power, symbolic execution engines have become more and more popular since the 2000s as useful test generation and correctness checker tools. The description of the recent trends and the solution for the challanges can be found in the survey of Baldoni et al. [2] presented in 2016.

KLEE [3] is a novel symbolic engine based on the LLVM assembly language. It behaves like a virtual machine and this way it can handle environmentally-intensive programs, for example by setting up a symbolic filesystem or by simulating faulty system calls. Symbolic PathFinder (SPF) [11] is developed at the NASA Ames Research Center as part of the Java PathFinder (JPF) verification toolchain [6]. It performs the symbolic interpretation on the Java bytecode. JPF and SPF are open-source projects; a symbolic executor created at our department, the so called Jpf Checker [7] was written by using it. SPF is highly customizable, for example it can be configured with several constraint solvers. One of them is the CORAL solver [15], which handles complex mathematical functions making it effective in scientific domains.

3. Background

3.1. Symbolic execution

The basic idea of symbolic execution is that the program is executed on symbolic input data instead of concrete values. During regular execution, the variables of the program have concrete values, meaning that the program follows a specific executional path determined by these values. A symbolic variable can hold any concrete value that is allowed for its type, so for example a symbolic integer can be an arbitrary whole number within the range of integer types. When the symbolic executor can not determine the exact value of a variable (because, for example, it is a user input or method parameter), a symbolic value is assigned to it. The possible values of symbolic variables can be bounded with constraints, which makes the symbolic execution more precise. These constraints are usually derived from conditional statements or assignments. If a statement contains symbolic variables the whole statement will be symbolic, and this applies for logical expressions too. A symbolic boolean can either be true or false, therefore, the symbolic engine will continue to execute both the true and false brances of a symbolic if statement. Consequently, in theory a symolic engine will explore every possible executional path and find hidden runtime exceptions. From the different executional paths, a directed, acyclic graph can be composed, which is called the symbolic execution tree (see Figure 1). Each node of the symbolic execution tree corresponds to distinct program states. For each program state, we can apply a logical formula called the *path condition* (PC). The PC is formed over the symbolic variables from the constraints derived from conditional statements and assignments.

Listing 1: Sample code

```
1  class Test{
2    private CustomType data;
3    public String getDataStr() {
4      if(data == null)
5        return null;
6      return data.toString();
7    }
8    public int getHashCode() {
9      int code = 0;
10     if(data != null)
11       code = getDataStr().hashCode();
12     return code;
13   }
14 }
```

Listing 1 shows a simplified portion of a Java class. Let us consider that the symbolic execution is started with the `getHashCode()` method starting in line 8. This function calculates a hash code regarding the value of a custom typed class member, `data`. The symbolic execution tree built up during the execution can be found in Figure 1. Figure 1 does not show the details of the `toString()` method

of the `CustomType`, and the execution of the `hashCode()` method is indicated with only a curved line. The path condition of each path is presented in transparent, black-bordered boxes next to the corresponding program states. For simplicity, it contains constraints only for the symbolic member variable, `data` and is presented only when there is a change, namely after conditional statements involving variable `data`. It can be seen that on the `false` branch of a symbolic `if` statement the negation of the logical expression is added to the PC. When investigating the source code and the execution tree it is obvious that if the `getDataStr()` is called from the `getHashCode()` function, the conditional statement in line 4 can never be true. Consequently, the program states colored with yellow are unreachable. This unreachability is expressed in the PC too: it is infeasible. A symbolic engine can detect unreachable states with constraint solvers. Constraint solvers are used to solve the PC, they can assign values to the symbolic variables that satisfy the logical formula. If an infeasible PC is found the execution will be stopped on that path, so the engine can avoid not only unnecessary execution but possible false positive warnings. As figure 1 indicates, if the execution was continued along the unreachable program state an incorrect *NullPointerException* would be reported.

3.2. RTEHunter

Our department[1] is developing SourceMeter [14], a static source code analyzer tool. It analyses C/C++, Java, C#, Python and RPG projects and calculates source code metrics, detects code clones and finds coding rule violations in the source code. RTEHunter (abbreviation of RunTimeException Hunter) is one of the static analyzers of the SourceMeter Java toolchain. It is designed to detect runtime exceptions in Java source code without actually executing the application in real-life environment. Currently it can detect four kinds of common failure: *NullPointerException*, *ArrayIndexOutOfBoundsException*, *NegativeArraySizeException*, and *DivideByZeroException*.

RTEHunter is a classical type symbolic executor engine written in C++. It performs seperate symbolic executions on each method found in the analysed Java source. The input for RTEHunter is the Abstract Semantic Graph (ASG) [5] of the analysed program, which is constructed by the SourceMeter. The ASG is a language-dependent representation of the source code that contains detailed semantic information in an internal graph representation. Using the ASG RTEHunter creates a language-independent Control Flow Graph (CFG) [1] for each method. Figure 2 shows the two CFGs composed for the methods of the sample code in Listing 1. Each method has its own CFG but they are interconnected with a call edge because the `getHashCode()` method uses the `getDataStr()` method.

The symbolic execution is performed by traversing through the CFGs. In a CFG each node represents a *basic block*. A basic block is a straight-line sequence of code with exactly one entry point and exaclty one exit point. It is guaranteed that no branching will happen during the execution of the program code represented by

[1]SourceMeter is developed at Department of Software Engineering, University of Szeged.

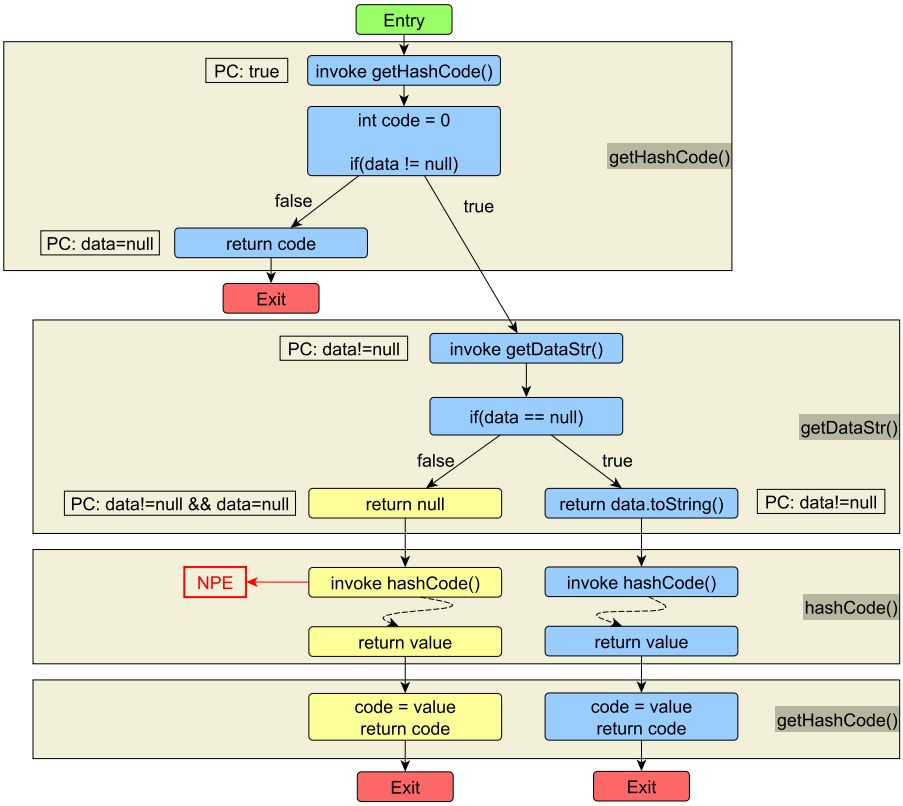

Figure 1: A simplified symbolic execution tree

a basic block, e.g. it does not include any jumps or jump targets. As it can be seen in Figure 2, every function call and conditional statement (amongst other language constructions) will start a new basic block. The entry point and the exit point of a CFG is represented by an EntryBlock (green box) and an ExitBlock (red block), respectively. The control can enter and leave the CFG of a method only through these nodes.

4. Contribution

The discussed example gives a quick insight on how the symbolic execution is preformed in the classical way. Both the presented source code and its conditional statements are simple, therefore, maintaning the path condition is easy. However, for more complex programs, the symbolic execution tree will be much larger. The number of branches grows almost exponentially with the number of symbolic conditional statements. This is what we call the path explosion problem. Due to the

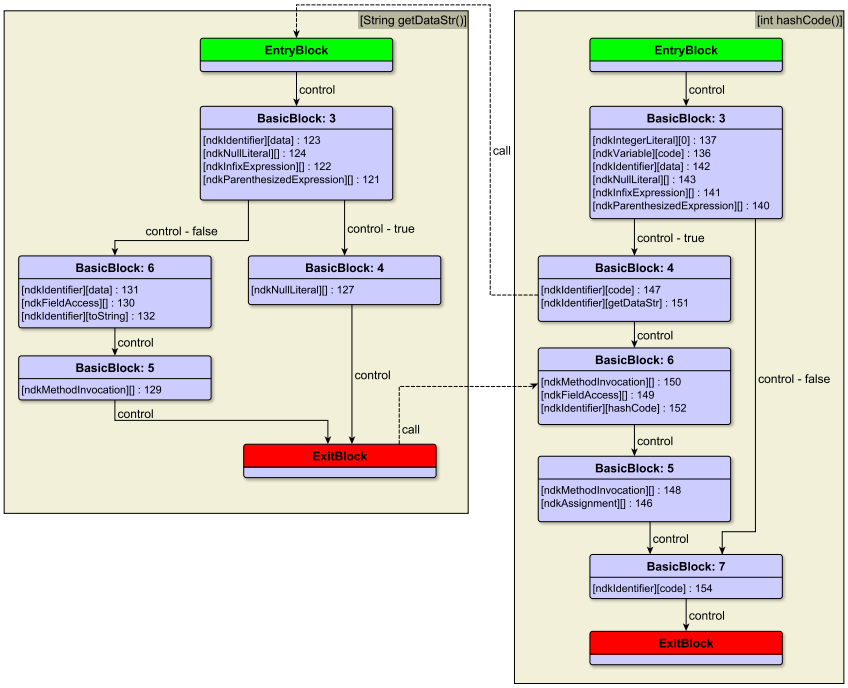

Figure 2: Control flow graph of the sample code shown in Listing 1

path explosion it is unlikely that symbolic engines can explore the whole symbolic execution tree exhaustively within a reasonable amount of time. Path explosion is in connection with the difficulty that we are addressing in this paper, namely the constraint solving problem. As previously mentioned, forming a path condition from the symbolic variables and checking its satisfiability at branching points are useful for pruning the symbolic exeecution tree, filtering out unreachable program states and hereby false positive warnings. In practice, constraint solvers suffer from many limitations that affect computational time significantly. They consume a big portion of the overall runtime causing symbolic engines to scale poorly on bigger systems. Constraint solving optimizations have to be made to maintain a trade-off between accuracy and scalability.

4.1. Null constraint solver

Themany optimization attempts are available in the current symbolic engines. One idea is to mix concrete and symbolic execution into the so called concolic execution. For example, if the constraint solver of the CUTE and jCUTE systems [13], [10] fails to satisfy a complex expression, the constraints are simplified by replacing some symbolic variables with concrete values. Another option is using incrementally

the results of similar constraint expressions. Ramos and Engler [12] introduced the under-constrainted symbolic execution whose basic idea – executing functions directly – is similar to the method by method approach implemented in RTEHunter.

Our team had gained experience in the use of constraint solvers with the predecessor of RTEHunter, as it was described in 2015 by Kádár et al. [8]. However, in RTEHunter we dismissed the introduced constraint solving mechanism. We decided to try out a more lightweight, heuristical approach instead of a conventional constraint solver and the result of this idea was the null constraint solver.

The basic idea of this simplified solver is that we store constraints refeering only to the null value of a symbolic variable. Constraints express only whether a symbolic object is null or not, therefore only null assignments and null value checks are considered during the build-up of the path condition. It is clear that the expressions will remain quite simple and checking the feasibility is also very easy. In Listing 1 on the `true` branch of the `if` statement in line 11 it will be noted that the member variable `data` is anything but null, so when the second conditional statement is reached in line 4 the infeasibility can be detected with almost zero effort. This approach did not alter notably the computational time requirements of RTEHunter whose previous version entirely lacked a constraint solving mechanism. We lost the arithmetic constraints, however this seemed to be a viable trade-off.

4.2. Results

We collected 209 open source Java systems to test the null constraint solver. These Java systems are from various industrial domains and their size ranges from a thousand to 2.5 million lines of code with the average of 140 000. To see the improvement, we executed both the original, constraint solverless and the improved versions of RTEHunter on the selected Java systems The results are presented in Table 1. As it can be seen there is a decrease in the number of *NullPointerExceptions* (NPE), *NegativeArraySizeExceptions* and *ArrayIndexOutOfBoundsExceptions*. The numbers hide that in fact 16 NPE were eliminitaed and 7 new were introduced. RTEHunter stops the execution along a path when a runtime failure is detected therefore it is obvious that the new warnings showed up because by eliminating unreachable paths the symbolic engine could explore other paths in the symolic executional tree before reaching the executional limits. The conclusion of the verification was that not only did we eliminate some serious false positive warnings, but also discovered 7 true positive runtime issues, though our heuristical approach needs future improvement.

RTEHunter	NullPointer-Exception	NegativeArray-SizeException	DivideBy-ZeroException	ArrayIndexOut-OfBoundsException
Original	3,102	24	167	681
Improved	3,093	23	167	675

Table 1: The results of the original and improved RTEHunter

5. Conclusions

Symbolic execution is a static analyser method that simulates the execution of a program by assuming symbolic values for inputs and environment dependent variables. Despite the fact that symbolic execution is a powerful tool for bug detection, there are many challenges and practical limitations. An important field of these difficulties is constraint solving which is useful for making symbolic engines more precise. Regardless of their positive effects, contraint solvers demand many resources, therefore finding more lightweight but still satisfactory solutions is an important challenge. We tried out a heuristical approach and tested it on more than 200 open source Java systems. The results showed that not only did it increase the accuracy of the symbolic engine by filtering out false positive warnings but more true positive runtime errors could be discovered within the same executional limits.

In the future, we plan to improve our heuristical constraint solving and find more solutions that would not increase the resource requirements significantly, but improve the efficiency of the algorithm.

References

[1] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *arXiv preprint arXiv:1610.00502*, 2016.

[3] Cristian Cadar, Daniel Dunbar, Dawson R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[4] P. David Coward. Symbolic Execution Systems – a Review. *Software Engineering Journal*, 3(6):229–239, November 1988.

[5] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 172–181. IEEE Computer Society, IEEE Computer Society, oct 2002.

[6] Java PathFinder Tool-set. `http://babelfish.arc.nasa.gov/trac/jpf`.

[7] István Kádár, Péter Hegedűs, and Rudolf Ferenc. Runtime exception detection in java programs using symbolic execution. *Acta Cybernetica*, 21(3):331–352, 2014.

[8] István Kádár, Péter Hegedűs, and Rudolf Ferenc. Adding constraint building mechanisms to a symbolic execution engine developed for detecting runtime errors. In *In Proceedings of the 15th International Conference on Computational Science and Its Applications*, pages 20–35, 2015.

[9] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[10] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings - International Conference on Software Engineering*, pages 416–425, 2007.

[11] Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[12] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, Washington, D.C., 2015. USENIX Association.

[13] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, Berlin, 2006. Springer-Verlag.

[14] The SourceMeter Homepage. `https://www.sourcemeter.com`.

[15] Matheus Souza, Mateus Borges, Marcelo D'Amorim, and Corina S. Păsăreanu. CORAL: Solving complex constraints for symbolic pathfinder. In *Lecture Notes in Computer Science*, volume 6617 LNCS, pages 359–374. Springer, Berlin, Heidelberg, 2011.