International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395 -0056
Volume: 03 Issue: 07 | July -2016 www.irjet.net p-ISSN: 2395-0072

A NOVEL SYMBOLIC EXECUTION MODEL IN AUTOMATED GENERATION OF TEST CASES

To Huu Nguyen[1*], Tran Thi Ngan[1], Do Thanh Mai[2], Tran Manh Tuan[1]
[1] *School of Information and Communication Technology, Thainguyen University, Vietnam*
[2] *School of Foreign Languages, Thainguyen University, Vietnam*

Abstract – *Modern software is becoming more and more complex and need to have a high reliability. A programmer can be assured that sample test runs work correctly by checking the results. Program testing is the most intensive tool for maintaining quality of software. It is executed on each input from a given set of inputs one by one. The biggest challenge of testing is how to make an effective set of inputs. Symbolic execution is a program analysis that is based on the same idea as testing. However, symbolic execution has a huge difference. Instead of supplying the normal inputs to a program, symbolic execution supplies symbols representing arbitrary values. In this paper, we introduce a novel symbolic execution model based on the combination of SPF with Choco decision procedure in solving complex string constraints. The proposed model is implemented on the examples that are used in other constraint solvers. The numerical statistics of the result is also given in this paper.*

Key Words: Test cases, Symbolic execution, automated generation of test cases, program testing, programing analysis.

1. INTRODUCTION

Software checking is the most necessary step in the completeness of computer programs. The objective of this step is to assure the reliability and the correctness of software. Two strategies for checking the correctness are testing and model checking [14]. Testing is often used but it is mostly performed manually [15], [21]. Moreover, testing meets the challenges at finding concurrent errors [3], [16]. This makes software testing be an expensive progress and still have low coverage of the source code. On the other hand, model checking is completely automatic and fast, frequently producing an answer in a matter of minutes [13], [17]. The main disadvantage of model checking is the state explosion problem [17], [24].

Symbolic execution is an analysis technique that generates high coverage test suite and also finds the deeply errors in complex software. The input values of symbolic execution are symbolic values instead of actual data. And its outputs are presented as a function of the symbolic inputs [7]. The ability to generate concrete test inputs is one of the major strengths of symbolic execution [6]. In [16], Sarfraz Khurshid et al. provided a two-fold generalization of symbolic execution and performed symbolic execution of code during explicit state model checking. The paper also illustrated two applications of their framework that are checking correctness of multi-threaded programs and generation of non-isomorphic test inputs.

Symbolic execution has been applied into generate test inputs for various goals. However, the most well-known use of this approach is to generate test inputs, to improve code coverage and expose software bugs [8], [10]. Besides, other uses of this approach include privacy preserving error reporting [9], automatic generation of security exploits [2], load testing [25], fault localization [20] regression testing, robustness testing and testing of graphical user interfaces, etc.

Two components of symbolic execution are path condition generation and path condition solving. A main challenge in symbolic execution is dealing with path conditions. To overcome this, a constraint solver namely CORAL is proposed [23] by M. Souza et al. CORAL supplies with a new constraint solver named heuristic solver and it also integrates this solver into SPF symbolic execution tool as well. But in this paper, CORAL is not evaluated in the context of constraints generation from the analysis of other applications yet. In [11], Indradeep Gosh et al. proposed an effective tool (JST) for the automated generation of test case with a high coverage. JST is a novel tool with a newly supported essential Java library components and widely used data structures. This tool supplies with new solving techniques mainly for string constraints, regular expressions. Moreover, it also supports to

integer and floating point numbers. Key optimizations make this tool be more efficient. However, the experimental results show that it is hard to reach to 100% code coverage for the sections under test since the deficiencies in the driver are manually generated.

2. BACKGROUND

2.1. Symbolic execution tools

In this section we want to introduce about briefly several recent tools that are based on symbolic execution.

Java PathFinder (JPF)

JPF [1] is considered as an explicit-state model checker for Java programs. It is built on top of a customized Java Virtual Machine (JVM). For convenient of users, JPF stores all the explored states. It always backtracks when it visits a previously explored state. The user can customize the search and it can specify what part of the state to be stored and used for matching. JPF focuses on finding bugs and uses a variety of scalability enhancing mechanisms.

Symbolic PathFinder (SPF)

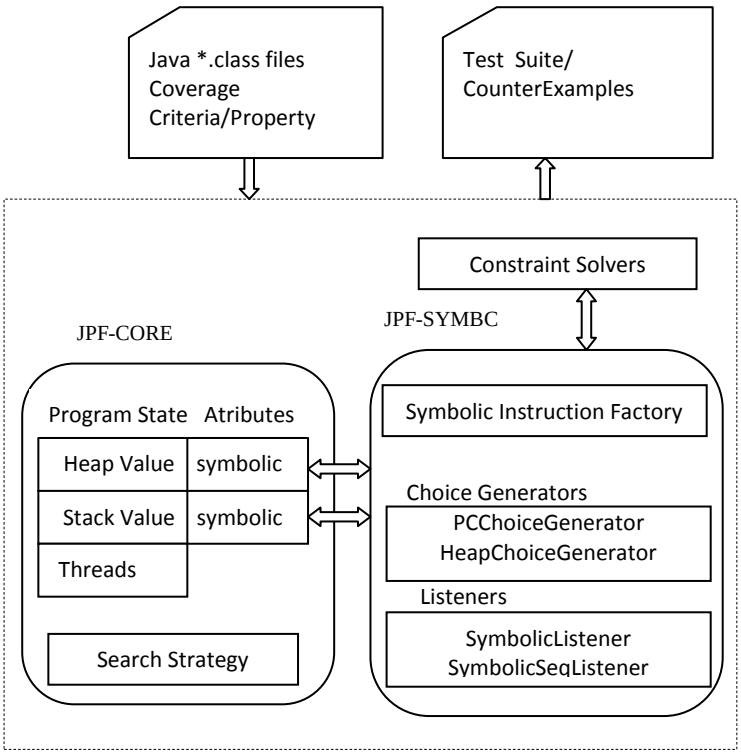

Fig – 1: Structure of SPF tool

SPF [18], [19] is part of the Java PathFinder verification tool-set. It is used to generate and explore the symbolic execution tree. It is also used to analyze thread inter-leavings and other forms of non- determinism that might be present in the code. The structure of SPF is given as in Figure 1 below.

DART (Directed Automated Random Testing)

DART [12] is based on an automated extraction of program interface from source code. It generates of test driver for random testing through interface. DART also supplies a dynamic test generation to direct executions along alternative program paths. Hence, DART can detect program crashes and assertion violations. DART deals with dynamic data easier with concrete executions. All bugs reported by DART are guaranteed to be sound. But it may not terminate and path space of a large program is huge. These are the main limitations of DART.

CUTE (A Concolic Unit Testing Engine)

CUTE [22] executes the code under test both concretely and symbolically at the same time. CUTE does not provide an automated extraction of interfaces. CUTE leaves it up to the user to specify which functions are related and what their preconditions are. Unlike DART that it was applied to testing each function in isolation and without preconditions, CUTE targets related functions with preconditions such as data structure implementations.

CUTE also uses backtracking to generate a test input that executes one given path. But it attempts to cover all feasible paths

2.2. Automated generation of test cases

EXE

EXE [4] is popular as an effective bug-finding tool that automatically generates inputs that crash real code. It uses a robust, bit-level accurate symbolic execution to find deep errors in code and automatically generate inputs that will hit these errors. EXE has advantages in modeling of memory and fast constraint solver. Based on these, EXE can perform an execution down any feasible program path and at dangerous operations. By using EXE, the test cases are generated with a high coverage and high ability in discovering deep bugs in a variety of complex code as well.

KLEE

KLEE [5] is a redesign of EXE, built on top of the Low Level Virtual Machine (LLVM) compiler infrastructure.

KLEE uses novel constraint solving optimizations that improve performance by over an order of magnitude and let it handle many programs that are completely

intractable otherwise. KLEE is an effective tool because of its space-efficient representation, its heuristic search and simple, straightforward approach to handling the environment. The automated generation of test suites with a high coverage on a diverse set of real, complicated, and environmentally-intensive programs is also the strength of KLEE. For evaluation purpose, KLEE is applied into all 90 programs in the latest stable version of GNU COREUTILS. In recent studies, KLEE is used in a variety of areas including wireless sensor networks, automated debugging, reverse engineering, testing of binary device drivers, exploit generation, online gaming and schedule memorization in multithreaded code.

CREST

CREST [6], [7] is a concolic testing tool for C programs. It generates test inputs automatically and executes target under test on generated test inputs. CREST can explore all possible execution paths of a target systematically. CREST is also an open-source re-implementation of CUTE. CREST's instrumentation is implemented as a module of C Intermediate Language (CIL). CREST has been used in various applications consisting of building tools for augmenting existing test suites to test newly-changed code, detecting SQL injection vulnerabilities, running distributed on a cluster for testing a flash storage platform and experimenting with more sophisticated concolic search heuristics.

3. A SYMBOLIC METHOD

In this section, we propose a framework that combines a symbolic execution and an automated generation of test inputs. The general diagram of this frame work is presented in Section 3.1. All the details about our work are shown in Section 3.2. In section 3.3, we address some main functions used in this model.

3.1. Modeling the novel frame work

In this section, we will introduce a new model (called NSE) based on the performance of SPF on string constraints. The general diagram of this model is presented as in Figure 2 below. The input of this model is a software program and the correctness specification. Firstly, the input program is transformed to the instrumented form by using code instrumentation. Consequently, this transformed program and correctness specification are used in SPF tool like necessary conditions to perform the model. The result of this step is counterexample or the test suite. Lastly, a generic decision procedure is built based on the status of the model through an interface. All the above steps are repeated until every branch of input program is browsed.

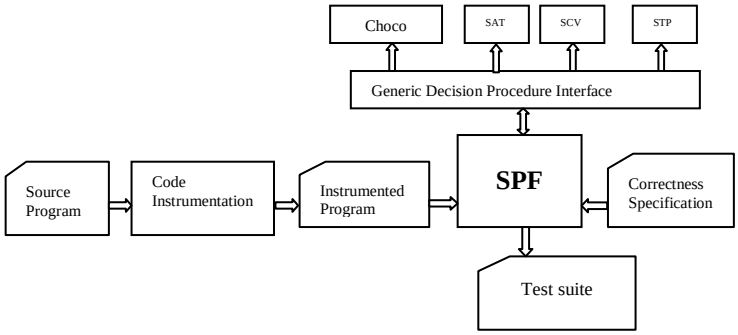

Fig - 2. General scheme of proposed model

3.2. Basic steps of proposed model

The steps of this model can be detailed as in Table 1 below.

Table – 1: Steps of novel framework

Input	Code program, Correctness specification
Output	Test cases and runtime, Error Report
NSE	
1	Transfer the input program to instrumented program
3	Repeat
2	Use SPF combining with Hybrid solver
3	Build Generic Decision Procedure Interfaces
4	Collect the constraints and solved through an SMT (Satisfiability Modulo Theories)
5	Until every branch of input program is tested
6	Error Report and Solved constraints to generate test suite.

4. EXPERIMENT RESULTS

4.1. Some basic functions

To illustrate for the using of SPF in NSE, especially in complex string solving, we apply the proposed model into examples. In which, these examples have been used in other solvers. We are going to show some basic functions of the model in this section. Figure 3 below

shows the function used to check the correctness of URL query chain.

```
package _StringTest;

public class Example {
    private static boolean IsEasyChairQuery(String str) {
        int lastSlash = str.lastIndexOf('/');
        if (lastSlash < 0) {
          return false;
        }
        String rest = str.substring(lastSlash + 1);
        if (!rest.contains("EasyChair")) {
          return false;
        }
        if (!str.startsWith("http://")) {
          return false;
        }
        String t =
            str.substring("http://".length(), lastSlash);
        if (t.startsWith("www.")) {

          t = t.substring("www.".length());
        }
        if (!(t.equals("live.com")) && !(t.equals("google.com"))) {
          return false;
        }
        return true;
      }
```

***Fig- 3:** Test the correctness of query chain.*

`PreserveSomeHtmlTagsAndRemoveWhitespaces(String body)` is applied to remove the spaces and simple HTML tag. It is presented as in Figure 4 below.

```
    public static String preserveSomeHtmlTagsAndRemoveWhitespaces(String body) {
        if (body == null)
            return body;
        int len = body.length();
        int i = 0;
        int old = i - 1;
        while (i < len) {
            if (i < old) {
                throw new RuntimeException("Infinite loop");
            }
            old = i;
            if (body.charAt(i) == '<') {
                if (i + 14 < len &&
                (body.charAt(i + 8) == '\"')&&
                (body.charAt(i + 7) == '=')&&
                (body.charAt(i + 6) == 'f' || body.charAt(i + 6) == 'F')&
                (body.charAt(i + 5) == 'e' || body.charAt(i + 5) == 'E')&&
                (body.charAt(i + 4) == 'r' || body.charAt(i + 4) == 'R')&&
                (body.charAt(i + 3) == 'h' || body.charAt(i + 3) == 'H')&&
                (body.charAt(i + 2) == ' ')&&
                (body.charAt(i + 1) == 'a' || body.charAt(i + 1) == 'A')){
                    int idx = i + 9;
                    int idx2 = body.indexOf("\"", idx);
                    int idxStart = body.indexOf('>', idx2);
                    int idxEnd = body.indexOf("</a>", idxStart);
                    if (idxEnd == -1)
                        idxEnd = body.indexOf("</A>", idxStart);
                    i = idxEnd + 4;
                    continue;
                }
            }
            i++;
        }
        return "";
    }
```

***Fig-4:** The code of String body function*

The purpose of function site_exec(String cmd) (in figure 5) in symbolic execution is to check the risk of query chains which can cause run-time exception errors.

```
    public static void site_exec(String cmd) {
        String PATH = "/home/ftp/bin";
        int sp = cmd.indexOf(' ');
        int j;
        String result;
        if (sp == -1) {
            j = cmd.lastIndexOf('/');
            result = cmd.substring(j);
        }
        else {
            j = cmd.lastIndexOf('/', sp);
            result = cmd.substring(j);
        }
        if (result.length()  + PATH.length()> 32) {
            System.out.println ("Will cause bufferoverflow");
            return;
        }
        String buf = PATH + result;
        if (buf.contains ("%n")){
            throw new RuntimeException ("String may contain threat");
        }
    }
    public static void main(String args[]){
        Example ex=new Example();
        ex.site_exec("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!%n");
        //ex.IsEasyChairQuery("http://www./EasyChair");
        //ex.preserveSomeHtmlTagsAndRemoveWhitespaces("<<<<<a href=\">    @");
    }

}
```

Fig-5: *site_exec(String cmd) function*

The proposed model - NSE has been implemented in Java program and executed on a PC HP Pavilion DV4 laptop with Core i3 processor, 4M RAM. The experimental results are obtained by using constraint solver "choco"(symbolic.dp=choco) and the decision procedure interface Automata (symbolic.string_dp=automata). The maximum computational time for string constraint solving is 3 seconds (timeoisymbolic.string_dp_timeout_ms=3000). Based on these functions, all the branches of the program are tested. The statistics of performance of the NSE model is given in Table 2.

Table – 2: Numerical statistics of the proposed model

Functions	String	Integer	Number of Iterations	PCs	Pre-processed	Timeouts
IsEasyChairQuery(String str)	3554	578	21	15	1	0
preserveSomeHtmlTagsAndRemoveWhitespaces (String body)	4513	1876	358	6148	1628	1258
site_exec (String cmd)	112	0	0	7	0	3

From table 2, the values on "String" column imply the sum of necessary time to solve the string constraints (mini-second). The values on "Integer" column give the

same information for the Integer constraint solving. "Number of Iterations" column shows the sum of number of necessary transference between string and integer solving. "PCs" column gives the sum of number of Path Constraints used in the model. "Pre-processed" column addresses the sum of path constraints found in order to define unsatisfied paths in preprocessing step. Finally, "timeouts" index is the number of unsolvable constraints that can be reached if the running time is more than 3 seconds.

5. CONCLUSIONS

Although the significant scalability challenge for symbolic execution is how to handle the exponential number of paths in the code. In the proposed model, this issue is completely solved by using SPF and Choco constraint solver in the complex mixed string constraints. The model is effective because it gives the test suite in which all the branches of the program are browsed in a limited running time. This paper also gives the experimented results obtained when applying the proposed model on the typical examples.

ACKNOWLEDGEMENT

The authors are grateful for the support from the staffs of Samsung ICTU Lab. The Lab has provided us the necessary devices as experimental tools. We want also to say thank TN2014-TN07-03, the project of Thainguyen University.

REFERENCES

[1]. S. Anand, C. S. Păsăreanu, and W. Visser, JPF-SE: a symbolic execution extension to Java PathFinder. In TACAS'07, pages 134–138, 2007

[2]. Brumley, D., Poosankam, P., Song, D. X., and 0002, J. Z., "Automatic patch based exploit generation is possible: Techniques and implications," in IEEE Symposium on Security and Privacy, pp. 143–157, 2008.

[3]. Borges, M., d'Amorim, M., Anand, S., Bushnell, D., & Pasareanu, C. S. (2012, April). Symbolic execution with interval solving and meta-heuristic search. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* (pp. 111-120). IEEE.

[4]. C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, EXE: Automatically generating inputs of death, In CCS'06, Oct–Nov 2006.

[5]. Cadar, C., Dunbar, D., & Engler, D. R. (2008), KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, In *OSDI* (Vol. 8, pp. 209-224).

[6]. Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C. S., Sen, K., Tillmann, N., & Visser, W. (2011, May). Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 1066-1071). ACM.

[7]. Cadar, C., & Sen, K. (2013). Symbolic execution for software testing: three decades later. *Communications of the ACM*, *56*(2), 82-90.

[8]. Cadar, C., Dunbar, D., & Engler, D. R. (2008, December). KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI* (Vol. 8, pp. 209-224).

[9]. Castro, M., Costa, M., and Martin, J.-P., "Better bug reporting with better privacy," in International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 319–328, 2008

[10]. Darringer, J. A., & King, J. C. (1978). Applications of symbolic execution to program testing. *Computer*, *11*(4), 51-60.

[11] Ghosh, I., Shafiei, N., Li, G., & Chiang, W. F. (2013), JST: an automatic test generation tool for industrial Java applications with strings. In*Proceedings of the 2013 International Conference on Software Engineering*(pp. 992-1001). IEEE Press.

[12]. P. Godefroid, N. Klarlund, and K. Sen, DART: Directed Automated Random Testing, In PLDI'05, June 2005.

[13]. Jhala, R., & Majumdar, R. (2009). Software model checking. *ACM Computing Surveys (CSUR)*, *41*(4), 21.

[14]. Jovanović, I. (2006). Software testing methods and techniques. *The IPSI BgD Transactions on Internet Research*, *30*.

[15]. Khan, M. E. (2010). Different forms of software testing techniques for finding errors. *International Journal of Computer Science Issues*, *7*(3), 11-16.

[16]. Khurshid, S., Păsăreanu, C. S., & Visser, W. (2003, April). Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 553-568). Springer Berlin Heidelberg.

[17]. Merz, S. (2001). Model checking: A tutorial overview. In *Modeling and verification of parallel processes* (pp. 3-38). Springer Berlin Heidelberg.

[18]. Păsăreanu, C. S., & Rungta, N. (2010), Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (pp. 179-180). ACM.

[19]. Păsăreanu, C. S., Visser, W., Bushnell, D., Geldenhuys, J., Mehlitz, P., & Rungta, N. (2013), Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis, *Automated Software Engineering*, *20*(3), 391-425.

[20]. Qi, D., Roychoudhury, A., Liang, Z., and Vaswani, K., "Darwin: An approach for debugging evolving programs," in Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 33–42, 2009.

[21]. Sawant, A. A., Bari, P. H., & Chawan, P. M. (2012). Software testing techniques and strategies. *International Journal of Engineering Research and Applications (IJERA)*, *2*(3), 980-986.

[22]. Sen, K., Marinov, D., & Agha, G. (2005). CUTE: a concolic unit testing engine for C. In *ACM SIGSOFT Software Engineering Notes* (Vol. 30, No. 5, pp. 263-272). ACM.

[23]. M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu, "CORAL: Solving Complex Constraints for Symbolic PathFinder," in NASA Formal Methods, 2011, pp. 359–374.

[24]. Yang, J., Twohey, P., Engler, D., & Musuvathi, M. (2006). Using model checking to find serious file system errors. *ACM Transactions on Computer Systems (TOCS)*, *24*(4), 393-423.

[25]. Zhang, P., Elbaum, S. G., and Dwyer, M. B., "Automatic generation of load tests," in International Conference on Automated Software Engineering, pp. 43–52, 2011

BIOGRAPHIES

Msc. To Huu Nguyen received the Bachelor Education of Information Technology at Thai Nguyen University of Education in 2003 and Master degree on Computer Science at Thainguyen University in 2008. He worked as a lecturer at Faculty of Information Technology, School of Information and Communication Technology, Thainguyen University from 2004. Now, he is a researcher at Institute of Information Technology, Academic Institute of Science and Technology, Vietnam.

Office address: University of Information and Communication Technology, Thai Nguyen University, Thai Nguyen, Vietnam.

Email: thnguyen@ictu.edu.vn.

Dr. Tran Thi Ngan obtained the Bachelor degrees on Mathematics- Informatics at VNU University of Science, Vietnam National University (VNU). She got Master degree on Computer Science at Thai Nguyen University. She received PhD degree on applied Mathematics – Informatics at Hanoi University of Science and Technology. From 2003, she worked as a lecture in Faculty of Information Technology, School of Information and Communication Technology, Thai Nguyen University. Her major interests are discrete mathematics, Monte Carlo method, optimization, probability theory and statistics, machine learning.

Msc. Do Thanh Mai received the Bachelor of Education in Information Technology at Thai Nguyen University of Education in 2003 and Master degree on Computer Science at Thai Nguyen University in 2008. She worked as a lecture of Information Technology in Basic Sciences Department at School of Foreign Languages (SFL-TNU).

Office address: School of Foreign Languages, Thai Nguyen University, Thai Nguyen, Vietnam.

Email: dothanhmai.sfl@tnu.edu.vn.

Msc. Tran Manh Tuan received the Bachelor on Applied Mathematics and Informatics at Hanoi University of Science and Technology in 2003 and Master degree on Computer Science at Thainguyen University in 2007. Now, he is a researcher at Institute of Information Technology, Academic Institute of Science and Technology, Vietnam. He worked as a lecturer at Faculty of Information Technology, School of Information and Communication Technology, Thainguyen University from 2003.
Office address: University of Information and Communication Technology, Thai Nguyen University, Quyet Thang, Thai Nguyen city, Thai Nguyen, Vietnam.
Email: tmtuan@ictu.edu.vn.