Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2016-IC-005

2016-IC-005

Symbolic Execution of Complex Program Driven by Machine Learning Based Constraint Solving

Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li

International Conference on Automated Software Engineering 2016

Most of the papers available from this document appear in print, and the corresponding copyright is held by the publisher. While the papers can be used for personal use, redistribution or reprinting for commercial purposes is prohibited.

Symbolic Execution of Complex Program Driven by Machine Learning Based Constraint Solving*

Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li
State Key Laboratory for Novel Software Technology
Department of Computer Science and Technology
Nanjing University, Nanjing, Jiangsu, P.R.China
{bulei|yuy}@nju.edu.cn

For Research Only

ABSTRACT

Symbolic execution is a widely-used program analysis technique. It collects and solves path conditions to guide the program traversing. However, due to the limitation of the current constraint solvers, it is difficult to apply symbolic execution on programs with complex path conditions, like nonlinear constraints and function calls.

In this paper, we propose a new symbolic execution tool MLB to handle such problem. Instead of relying on the classical constraint solving, in MLB, the feasibility problems of the path conditions are transformed into optimization problems, by minimizing some dissatisfaction degree. The optimization problems are then handled by the underlying optimization solver through machine learning guided sampling and validation. MLB is implemented on the basis of Symbolic PathFinder and encodes not only the simple linear path conditions, but also nonlinear arithmetic operations, and even black-box function calls of library methods, into symbolic path conditions. Experiment results show that MLB can achieve much better coverage on complex real-world programs.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Symbolic Execution; Machine Learning; Complicated Path Condition; Constraint Solving

*This work is supported by the Joint NSFC-ISF Research Program, jointly funded by the National Natural Science Foundation of China and the Israel Science Foundation (No.61561146394), the National Natural Science Foundation of China (No. 61572249, No. 61472179, No.91418204), Jiangsu Science Foundation (BK20160066), and the 2015 Microsoft Research Asia Collaborative Research Program.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970364

1. INTRODUCTION

Symbolic execution [3] is a widely employed program analysis technique which executes programs with symbolic inputs. In the process of symbolic execution, a path condition, which is a conjunction of symbolic constraints, is maintained and gets updated whenever the program executes a branch instruction. Concrete inputs triggering the corresponding path can then be generated by constraint solving [5, 25].

The processing capability of symbolic execution is heavily relied on the underlying constraint solver. It is well known that even strong SMT solvers like iSAT [8] and Z3 [6] can not handle complex nonlinear constraints well [1], not to mention the black-box library methods or even native methods in real case codes.

The simple example shown in Figure 1 can be a typical representative of such programs. To cover the path of line 4, 5 and 6, symbolic execution needs to find a solution that satisfies the corresponding if statements in line 4 $(z > x)$ and line 5 $(x + y/(y + 2) == x * x - y)$. However, the nonlinear arithmetic operation $(x + y/(y + 2) == x * x - y)$ may be difficult for most of the existing solvers. Besides, the calculation of z which needs the function call $Long.numberOfLeadingZeros(x)$ is also a great challenge since normal constraint solvers have no idea to deal with the uninterpreted item.

```
1 public class program{
2  public static void example(long x, double y){
3     double z = Long.numberOfLeadingZeros(x);//library method
4     if(z>x)
5        if(x+y/(y+2) == x*x −y) //nonliner arithmetic operation
6           assert(false);
7  }
8  public static void main(String[ ] args){
9     long parm1 = Long.parseLong(args[0]);
10    double parm2 = Double.parseDouble(args[1]);
11    example(parm1,parm2);
12 }
13 }
```

Figure 1: A Simple Java Program with Complicated Path Conditions

To handle such programs with complicated constraints, intensive investigations have been conducted. Concolic testing [21] tries to go through the complex path conditions by replacing complicated symbolic terms of them with concrete values. Unfortunately, such simplification also decreases the probability to find more desired solutions. Heuristic search [24, 7] is a new trend. It finds the solutions that sat-

isfy the constraints with the help of heuristics like genetic algorithms (GA) [9], tabu-search [11], etc. These approaches perform well in dealing with certain problems, revealing the potential of such direct search methods. However, the capacity of these heuristic methods is little understood, and the configurations of these approaches are hard to set to achieve stable performance.

In this paper, we present a new symbolic execution tool, MLB, which is driven by *M*achine *L*earning *B*ased constraint solving, to handle the complicate programs discussed above. First, MLB encodes all the difficult operations, including nonlinear constraints, mathematical methods (e.g. sin, log), floating point symbolic variables and the library method calls, as (uninterpreted) symbolic constraints.

Then, different from existing tools, MLB transforms the feasibility problems of the path conditions into optimization problems. A recently introduced machine learning based optimization technique, RACOS [26], is adopted to solve the optimization problems. RACOS has a solid theoretical foundation for complex optimization problems. It basically samples solutions from a learning model, updates the model from the feedback of the solutions, and continues this iteration to find better solutions. MLB adapts RACOS by using the dissatisfaction degree as the feedback of the sampled solutions, and thus RACOS converges to feasible path conditions as quickly as possible.

MLB is implemented on the basis of Java symbolic execution engine, Symbolic PathFinder [16, 18]. To evaluate the performance of MLB, intensive case studies are conducted to generate test cases for 16 real case programs, with a total number of 290 methods full of nonlinear operations, floating point arithmetic and even native method calls. The experimental results prove that our tool outperforms the other state-of-the-art tools in terms of both effectiveness and efficiency. Furthermore, it is worth to note that, unlike most of the existing tools, MLB is fully automatic that users do not have to modify any single line of their code to use our tool which means great convenience.

The rest part of this paper is organized as follows. The background of machine learning based constraint solving is introduced in section 2. Section 3 introduces the symbolic execution architecture of MLB and presents how MLB encodes complex program behaviors as symbolic constraints and solves them by machine learning. The implementation of MLB and the intensive case studies are introduced in section 4. The related works are discussed in section 5. Section 6 is the conclusion of the paper.

2. MACHINE LEARNING BASED OPTIMIZATION METHOD

As mentioned above, the major obstacle in applying symbolic execution to real world programs is the capability of constraint solving which targets of finding solutions to make all constraints satisfiable. The problem is closely related with the optimization problem: finding solutions to minimize the dissatisfaction degree.

However, the optimization problem for general complex constraints is usually quite complex too. One kind of solving techniques suitable for complex optimization tasks are derivative-free methods. They use only the information of solution evaluations instead of the objective function gradients, and thus are not misled by, e.g., gradients on complex optimization tasks. They commonly consist of an iteration of two key steps: sample solutions from a model (initially the uniform sampling from the solution space), and update the model from the objective function values of the sampled solutions. Many derivative-free methods have been proposed (e.g., [15, 2, 12, 14]). Unfortunately, most of these methods have little theoretical foundation due to the difficulty of analysis. Very recently, a machine learning based optimization method named RACOS was developed with solid theoretical ground [26], of which the required number of samples is well upper bounded in the approximation quality of the final solution.

This new machine learning based algorithm employs a hyper-rectangle classifier as its model. Better solutions can be distinguished from the worse ones by training the classifier according to the objective function values of the sampled solutions through covering all better solutions within the hyper-rectangle. The classifier, once been trained, produces a hyper-rectangle in the solution space, which indicates the region that contains better solutions. The solutions for the next iteration are then sampled from the hyper-rectangle. The simplicity of this hyper-rectangle classifier model allows the algorithm to be well analyzed. The analysis indicates that, to achieve a high optimization efficiency, the classifier should be highly randomized and the region of better solutions should be small [26]. The new algorithm RACOS [26] is accordingly designed.

This new machine learning based algorithm has remarkable merits comparing with other derivative-free optimization methods. Besides its solid theoretical foundation, it has also been empirically verified to have high efficacy and high efficiency [26], particularly in high dimensions ranged up to thousands, with default algorithm parameters. It supports optimization tasks with continuous, discrete [20], and mixed variables, and it almost has no parameters to adjust.

3. TECHNIQUE UNDER MLB

To overcome the limitations of the classical symbolic execution, MLB adapts the classical symbolic execution framework with a new sampling-validation style machine learning based solving as reviewed in section 2. The main workflow of MLB, Figure 2, is as follows:

- The underlying machine learning solver proposes a valuation sample.
- The upper layer symbolic execution engine validates the guessed sample by direct execution.
- The solver converges to the correct answer by analyzing the validation feedback using machine learning.

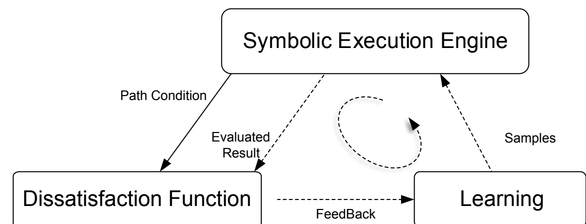

Figure 2: Machine Learning Based Solving under MLB

For Research Only

In order to fulfill the workflow and take advantage of the new machine learning based optimization technique RACOS to handle more complex programs, two main techniques underlying MLB are presented below:

- Transform the feasibility problem of the path condition to optimization problem, to take advantage of the machine learning based method.
- Enrich the path condition by encoding library method calls as uninterpreted path conditions which can be handled in the sampling-validation style.

3.1 Dissatisfaction Function Oriented Path Condition Optimization

To apply the machine learning based optimization technique RACOS, MLB transforms the feasibility problem of a path condition into an optimization problem through the *dissatisfaction function* so that a sampling-validation style solving can be conducted.

Given a path condition with the conjunction of n constraints C_i $(1 \leq i \leq n)$, we define a *dissatisfaction degree* D_i for each C_i. D_i is used to measure how badly a sample violates the constraint C_i. For example, the most straightforward way for defining D_i is that if C_i is satisfiable by a valuation sample S, then $D_i = 0$, otherwise, $D_i = 1$. Then, we can generate a *Dissatisfaction Function* $Df = \Sigma_{i=1}^{n} D_i$ to measure the total distance from the evaluated valuation sample S to a correct solution of the path condition.

To accelerate the convergence speed, MLB calculates the dissatisfaction degree as the formula shown below where op_i is the comparator of C_i and $left_i$ stands for the left expression value of it while $right_i$ is the right expression value. In detail,

$$D_i = \begin{cases} 0, & \text{if S satisfies } C_i \\ 1, & \text{if S dissatisfies } C_i \text{ and } op_i \text{ is } \neq \\ |left_i \text{ - } right_i|, & \text{if S dissatisfies } C_i \text{ and } op_i \text{ is not } \neq \end{cases}$$

Clearly, when the dissatisfaction function result is 0, the path condition is satisfied. In this manner, we transform the feasibility problem of the path condition into an optimization problem to minimize the dissatisfaction function, so that the machine learning based optimization solving method reviewed in section 2 can be used to analyze the complex path conditions in MLB.

3.2 Library Call Related Path Condition Generation

To support the diverse and complex codes in the real world, MLB generates more comprehensive path conditions during program analysis. The details of the path conditions enhanced in MLB are described as follows.

In the classical symbolic execution, the handling of third-party library method is always a big challenge. The reason is that it is difficult to present the third-party library method calls as simple mathematical constraints since the contents of the library methods are often uninterpreted.

Luckily, in our machine learning style solving, the specific contents are not indispensable. Actually, all the constraints in the optimization program can be handled in a black-box style. The symbolic execution engine can validate them by executing the corresponding statements with the sample guessed directly without knowing the deep detail.

Therefore, in MLB, each library method call is presented as an uninterpreted constraint, where the function name is treated as the uninterpreted operator in the constraint and the input arguments of the function call consistitue the symbolic variables in this library method PC.

Take the code in Figure 1 for example, statement 4 ($z > x$) needs a function call $Long.numberOfLeadingZeros(x)$. MLB will generate a library method PC $Long.numberOf-LeadingZeros(x) > x$ for it. Suppose the sample is $(x = 2)$, we call the method with argument $(x = 2)$ directly and get the return value 62. Then the result is used to calculate the dissatisfaction degree value.

Actually, all the function calls in the program can be encoded as uninterpreted symbolic constraints and processed in such a black-box manner. As a result, MLB can provide users black-box options to mark specific functions as black-box. Then MLB will avoid going through the detail of the specific functions and increase the efficiency.

Besides of the library call related PC, another special class of PC, domain related PC, is generated in MLB. The main idea is that samples proposed by the underlying solver may violate the path condition's domain requirement and cause exceptions. Therefore, MLB analyzes the basic constraint first, and adds special constraints to make sure the proposed sample does not violate the domain requirement. For example, MLB generates $(y \neq -2)$ for $(x+y/(y+2) == x*x-y)$ in statement 5, Figure 1. Due to the space limitation, the detail of domain related PC generation is omitted here.

4. IMPLEMENTATION AND EVALUATION

This section presents the implementation and performance evaluation of MLB. The implementation and all the data used in the evaluation are available from GitHub, https://github.com/MLB-SE.

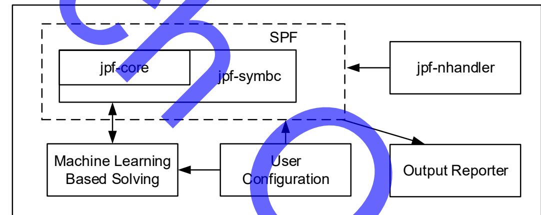

Figure 3: MLB Tool Architecture

4.1 Implementation

MLB is implemented on the basis of Symbolic PathFinder (SPF) [16, 18], which is a Java symbolic execution engine based on Java PathFinder (JPF) [13]. The architecture of MLB is shown in Fig.3.

• The classical program traversing core in SPF provides symbolic path condition generation for the execution semantics. MLB adapts it with more comprehensive path condition generation, including the library method call related PC and domain related PC as mentioned in section 3.

•To make MLB easier to use, we intergrate jpf-nhandler [23] into MLB to execute the complex path conditions with native methods more efficiently. The jpf-nhandler module delegates the execution of the native methods as it provides

For Research Only

the necessary information of them for MLB to generate the uninterpreted constraints.

• To solve the collected path conditions, the classical constraint solving is replaced by a new sampling and validation style machine learning based constraint solving module. A generic interface is created for the frequent interaction between the symbolic execution core and the machine learning based solver. Information required during the sampling and validation iterations is transmitted through the interface so that valid solutions can be quickly found and referred to the exploring process.

•Additionally, we employed and updated the convenient publisher system of SPF to provide the program analysis data and produce test cases from the solutions. Users can fetch these data easily and analyze them with the help of other off-the-shelf tools freely. For example, following the merits of [7], MLB supports the automatic invoking of JaCoCo[1] coverage measuring library to demonstrate the coverage reached by the generated test cases.

4.2 Tool Usage

```
1. target = program                                  //Target class under test
2. classpath = <project-main-class>    //File path of the target class
3. symbolic.method = program.example(sym#sym)
                                                   //Target method under test
4. symbolic.dp = CSP                               //Default solver setting
5. symbolic.mlpm = 4000                              //Default sample size
6. @using = jpf-nhandler                                //Call jpf-nhandler
7. nhandler.delegateUnhandledNative = true        //Call jpf-nhandler
8. nhandler.spec.skip = java.lang.Long.numberOfLeadingZeros
                        //Set the specific function in the black-box mode
```

Figure 4: Configuration of MLB for the Example in Fig.1

Except the special capability of handling complex programs with complicated constraints and library method calls, another key feature of MLB is the user-friendliness. MLB is a fully automatic symbolic execution engine for Java. Unlike most of the existing tools, users only need to configure our tool expediently before analysing their codes. MLB does not require users to modify any single line of their codes.

Figure 4 presents an illustration of the configuration for the code shown in Figure 1 to use MLB. As MLB is implemented on the basis of SPF, it shares the configuration style of SPF. Line 1 - 3 in Figure 4 show the necessary settings required by SPF for declaring the target. It points out the target under test is the method *example* in class *program*.

Line 4 sets the constraint solver under MLB to CSP which implements RACOS algorithm [26]. Line 5 is the optional parameter for MLB to set the sample size of each round for CSP to 4000. In another word, CSP guesses 4000 valuations each round and presents them to MLB to validate. If users do not set the parameter, the default value is 3000. It is worth to note that this is the *only* parameter that relates to the performance of the underlying machine learning solver.

For running the specific function call *java.lang.Long.numberOfLeadingZeros*, line 6 - 8 are added to call the extension jpf-nhandler. More specifically, line 8 tells MLB to encode and analyze the function as an uninterpreted function in a black-box mode.

[1]http://www.eclemma.org/jacoco/

Table 1: Benchmark for the Experiments

Program	*Operations*	*Method*	*Loc*	*From*
coral	Trigonometric functions, logarithms, polynomials	86	254	[7]
dart	Polynomials, required overflow	5	11	[7]
hash	Polynomial, shift, bit-wise xor	7	34	[7]
opti	Exponentials, square roots	8	24	[7]
power	Exponential function	3	20	[7]
ray	Polynomials (dot product)	35	190	[7]
sine	Float to bit-vector conversion	7	184	[7]
stat	Mean and std. dev. computation	17	61	[7]
tcas	Constant equality checks	13	75	[7]
tsafe	Trigonometric functions	8	63	[7]
airy	Polynomials, square roots, logarithms	11	296	[19]
bess	Polynomials, square roots	19	191	[19]
caldat	Polynomials, trigonometric functions	6	86	[19]
ell	Polynomials, square roots, trigonometric functions	31	484	[19]
gam	Logarithms, factorials, exponentials	21	195	[19]
ran	Polynomials, exponentials, xor, logarithms, square roots	13	218	[19]

Furthermore, as mentioned before, MLB also provides template script files, following the format of [7], to automatically run the test cases and invoke JaCoCo to generates visualized coverage reports. With the help of the script files and the configuration files, users can get the detail coverage report with a push button style user experience. More details can be found on the GitHub link mentioned before.

4.3 Evaluation Setup

To evaluate the performance of MLB, a variety of programs are used mainly from 2 sets of real case benchmarks. The first set of programs is from study [7], consisting of different kinds of nonlinear operations in real tool distributions[2]. The second set of the benchmarks is selected from the classical numerical computation book *Numerical Recipes* [19]. Typical pointer-free scientific computing functions whose constraints contain nonlinear computation and function calls are chosen as benchmarks and divided into 6 programs in accordance of the structure of the book. The basic information of the benchmarks is listed in Table.1.

In our experiments, MLB and 4 state-of-the-art symbolic execution tools are used to conduct test case generation on all the benchmarks. Then, we use the JaCoCo coverage measuring library to measure the instruction, branch and line coverage achieved by every competitors. Each tool is repeated for 5 times to eliminate the randomness. To prevent small programs from dominating the mean coverage of the tools, we weight each program's contribution with different metrics when computing the arithmetic mean on this specific metric.

The competitors used in the experiments including: jCUTE [22], SPF-Mixed [17], SPF-CORAL [24] and Con-

[2]Please refer to [7] for the detail introduction of these programs. Also note that the line of codes(Loc) listed in Table 1 is different from the Table 1 in [7], as that the Loc values reported here are counted by JaCoCo, where useless lines like blank lines are not counted as valid lines by JaCoCo.

For Research Only

colic Walk (CW) [7]. These tools stand for different classes of symbolic execution methods, for example, concolic testing, heuristic searching and so on. All the experiments are conducted on a desktop running Ubuntu 12.04 LTS with 3.10GHz Intel Core i5 and 4GB RAM, with time limit 300 seconds per method.

4.4 Experimental Study

MLB's performance is evaluated on the aspects of both effectiveness and efficiency as follows.

• Effectiveness Analysis

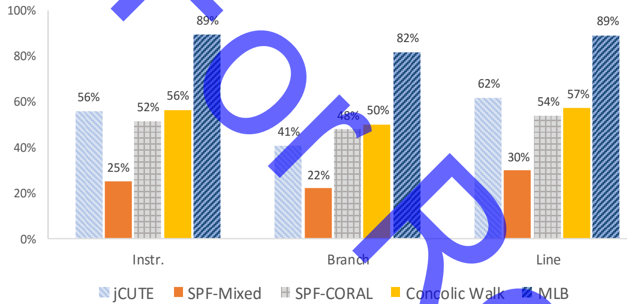

Figure 5: Weighted Coverage Reports

As described above, weighted coverage is used as the main indicator of effectiveness analysis. The data on all the programs with respect to different metrics is summarized in Figure 5. While computing the arithmetic mean over all the programs, the respective metric weight factor is introduced to prevent small benchmarks dominating the coverage. For example, for line coverage, we use the line of codes as weight, while for branch coverage the number of branches is treated as the weight.

It is clear that MLB outperforms all the other competitors significantly on all the coverage metrics. As seen from Figure 5, MLB's instruction coverage (89%) ranges from 1.6 times of CW (56%) to 3.6 times of SPF-Mixed (25%). This set of experiments strengthens our belief that by the help of machine learning based solving technique, MLB can handle the symbolic execution of complex nonlinear programs effectively.

• Efficiency Analysis

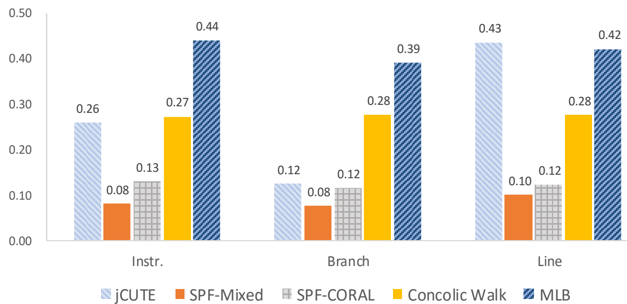

Figure 6: Weighted Efficiency Reports

The second aspect evaluated in the experiment is efficiency. To evaluate the efficiency of MLB, we use the coverage including instruction, branch and line achieved per unit of generation time as the metrics. Similar with Figure 5, we compute the average efficiency over all the benchmarks of every tool and summary the results in Figure 6.

Overall, MLB achieves better performance than other competitors. In case of instruction efficiency, MLB (0.44%/s) ranges from 1.6 times as efficient as CW (0.27%/s) to 5.5 times of SPF-Mixed (0.08%/s). The branch efficiency shares the similar results. For line efficiency, MLB is more efficient than other 4 tools except for jCUTE. Though jCUTE (0.43%/s) is as efficient as MLB (0.42%/s), MLB achieves much higher line coverage than jCUTE (89% vs 62%).

• Processing of Library Method Call

As described in section 3, MLB is able to handle library method calls in a black-box mode by encoding them as uninterpreted constraints.

To assess this special capability of MLB, we replace the mathematical methods in *coral* with the methods in the widely-used mathematics and statistics library *The Apache Commons Math*[3]. Then, we use MLB to generate test cases for the modified *coral* in the black-box mode. As Symbolic PathFinder can load the library method calls and execute the instructions one by one, we conduct the classical white-box analysis on this program as well.

The result is quite interesting. MLB achieves 79% branch coverage in 345.6 seconds with black-box mode. However, if we run MLB in the classical white-box mode, in another word, traversing each line of instructions in the library method, the branch coverage drops to 62% with even much longer time, 19347 seconds.

The reason is that in the classical white-box mode, we have to go through the complete structure of the methods called step by step. While in the black-box mode, we compact the third-party function calls as uninterpreted symbolic constraints and solve them directly. Thus, we can put the resource in the traversing of the main program itself to achieve higher coverage.

5. RELATED WORK

In the areas of software engineering, symbolic execution tools including KLEE [4], SPF[16, 18], Pex [25] and so on play an important role in many program analysis cases. It is well recognized that the applicability and scalability of these tools rely heavily on the underlying constraint solver. However, it is difficult for existing constraint solvers like Z3 [6] to handle complex path conditions involving nonlinear operations and library method calls, which appear commonly in real-world programs.

For solving those complex path conditions, several mitigation strategies are proposed. As one typical representative of simplification-based approaches, concolic testing [21] which has been used in jCUTE [22] and SPF-Mixed [17] tries to handle complex path conditions by replacing complex terms with concrete values. Besides, there are also other approaches which substitute nonlinear constraints by linear envelopes, e.g. [1], to abstract the state space and make it solvable by linear constraint solvers. Unlike the simplification-based solving with the abstraction of the search space, MLB encodes all the complex behaviors as they are and guides the solver to converge to the correct answer efficiently.

As used in CORAL solver [24] and CW algorithm [7], search-based approaches perform symbolic execution in a heuristic search style. For example, particle-swarm opti-

[3]https://commons.apache.org/

For Research Only

mization [10] and tabu search [11] are used to solve complex path conditions in the above works. However, these heuristic search algorithms are commonly weak in their theoretical foundation, which blocks the rational understandings of their optimization performance. In contrast, the machine learning based algorithm adapted in MLB is theoretically grounded ensuring the stable and great performance of the tool.

Last but not least, on the implementation level, most of the existing tools, e.g. [7], require users to modify their code according to certain regulations firstly before starting the analysis. While in MLB, users can feed their code to the tool directly, which improves the ease of use significantly.

6. CONCLUSIONS

In this paper, we propose a new Java symbolic execution tool, MLB, which is supported by a machine learning based optimization solving technique. Different from existing works, the solver under MLB works with the symbolic execution core in a sampling-validation and learning style interaction loop. In this manner, MLB supports the symbolic execution of not only simple codes with linear path conditions, but also complex real world programs with complicated nonlinear constraints and calls of library methods.

MLB is an automatic tool implemented on the basis of SPF, jpf-nhandler and JaCoCo. Therefore, MLB can be used to analysis complex Java codes with library methods and can get the coverage report directly without changing any line of the code. An intensive set of case studies on real-world case programs shows that MLB supports a wide range of well-known difficult real-world programs with outstanding efficacy and efficiency.

7. REFERENCES

[1] Barr, E. T., Vo, T., Le, V., and Su, Z. Automatic detection of floating-point exceptions. *ACM SIGPLAN Notices 48*, 1 (2013), 549–560.

[2] Beyer, H., and Schwefel, H. Evolution strategies - A comprehensive introduction. *Natural Computing 1*, 1 (2002), 3–52.

[3] Boyer, R. S., Elspas, B., and Levitt, K. N. SELECT-a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices 10*, 6 (1975), 234–245.

[4] Cadar, C., Dunbar, D., Engler, D. R., et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008), vol. 8, pp. 209–224.

[5] Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C. S., Sen, K., Tillmann, N., and Visser, W. Symbolic execution for software testing in practice: preliminary assessment. In *ICSE* (2011), ACM, pp. 1066–1071.

[6] De Moura, L., and Rner, N. Z3: an efficient SMT solver. In *Theory and Practice of Software, TACAS* (2008).

[7] Dinges, P., and Agha, G. Solving complex path conditions through heuristic search on induced polytopes. In *FSE* (2014), ACM, pp. 425–436.

[8] Franzle, M., Herde, C., Teige, T., Ratschan, S., and Schubert, T. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation 1* (2007), 209–236.

[9] Galeotti, J. P., Fraser, G., and Arcuri, A. Improving search-based test suite generation with dynamic symbolic execution. In *ISSRE* (2013), IEEE, pp. 360–369.

[10] Gies, D., and Rahmat-samii, Y. Particle swarm optimization (PSO) for reflector antenna shaping. In *APS/URSI* (2004), vol. 3, IEEE, pp. 2289–2292.

[11] Glover, F. Tabu search: A tutorial. *Interfaces 20*, 4 (1990), 74–94.

[12] Goldberg, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[13] Havelund, K., and Pressburger, T. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer 2*, 4 (2000), 366–381.

[14] Kennedy, J., and Eberhart, R. *Swarm Intelligence*. Morgan Kaufmann, 2001.

[15] Kirkpatrick, S., Gelatt Jr, C. D., and Vecchi, M. P. Optimization by simulated annealing. *Science 220*, 4598 (1983), 671–680.

[16] Păsăreanu, C. S., and Rungta, N. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE* (2010), ACM, pp. 179–180.

[17] Păsăreanu, C. S., Rungta, N., and Visser, W. Symbolic execution with mixed concrete-symbolic solving. In *ISSTA* (2011), ACM, pp. 34–44.

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

[19] Press, W. H. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[20] Qian, H., and Yu, Y. On sampling-and-classification optimization in discrete domains. In *CEC* (2016).

[21] Sen, K. Concolic testing. In *ASE* (2007), ACM, pp. 571–572.

[22] Sen, K., and Agha, G. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification* (2006), Springer, pp. 419–423.

[23] Shafiei, N., and Breugel, F. v. Automatic handling of native methods in Java PathFinder. In *SPIN* (2014), ACM, pp. 97–100.

[24] Souza, M., Borges, M., d'Amorim, M., and Păsăreanu, C. S. CORAL: solving complex constraints for Symbolic PathFinder. In *NASA Formal Methods*. Springer, 2011, pp. 359–374.

[25] Tillmann, N., and De Halleux, J. Pex–white box test generation for. net. In *Tests and Proofs*. Springer, 2008, pp. 134–153.

[26] Yu, Y., Qian, H., and Hu, Y.-Q. Derivative-free optimization via classification. In *AAAI* (2016), pp. 2286–2292.

For Research Only