International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 07 Issue: 06 | June 2020 www.irjet.net p-ISSN: 2395-0072

Finding New Integer Constraints Incorporated with String Constraints using Automata in Symbolic Execution

To Huu Nguyen[1], Tran Thi Ngan[2*]

[1]*University of Information and Communication Technology, Thai Nguyen, Vietnam*
[2]*Faculty of Computer Science and Engineering, Thuyloi University, 175 Tay Son, Dong Da, Hanoi, Vietnam;*

Abstract – *In computer programs, the string data appears regularly, especially in the part of programs or methods. Java language providing a large library of string data types but still contains many bugs. In software testing, we are interested in identifying bugs. In symbolic execution, string constraints are also considered in most of cases. In this research, we propose replacing the entire executable representation of an integer with specific integer values and determining by best estimating the integer constraint solver. Forces the string to the specified integer value then. If those integer values do not give the content string constraint satisfactory, we add the negative integer constraints with the replaced value to the integer constraint. Other specific values are generated continuously until a satisfactory string value is obtained or it is impossible to predict further specific integer values. We conclude that the string constraint does not exist satisfying value then.*

Key Words: String constraints, integer constraint, constraint solving, Java Path Finder, Automata.

1. INTRODUCTION

Any software needs to be tested before broadcasting. Many security and effective testing applications have been introduced. The purpose of these applications is to check whether certain properties of a program satisfy any possible usage scenario. Symbolic execution and concrete execution are the most common strategies in software testing. Symbolic execution was proposed by King *et al.* [9]. Symbolic execution is a program analysis method used to execute a program by symbolic values. The property was considered in most cases in the execution. Symbolic execution often explores multiple paths that a program could take under different inputs. This means that the checked property of a program is guaranteed [3]. Apart from the beneficial characteristics, symbolic execution also has some challenges such as

- Memory: symbolic execution may causes the path exploration. The memory could rises rapidly from handling pointers, arrays, or other complex objects.
- Environment: the interactions across the software stack also need to be cared in testing process.
- State space explosion: using symbolic input instead of concrete common input makes the path exploration. This leads to the difficulty in solving the state space exploration as well.
- Constraint solving: how can constraint solver work on a software testing? In the testing progress, it maybe have to deal with a huge amount of constraints.

Here is an example of symbolic execution [1]:

```
1 int intExp(int a,int n) {
2  if (n < 0)
3    throw new ArithmeticException();
4  else {
5    int out = 1;
6    while (n > 0) {
7      out = out*a;
8      n--;
9    }
10   return out;
11 }
12 }
```

Fig-1. The source code

The result of symbolic execution is formed as a tree:

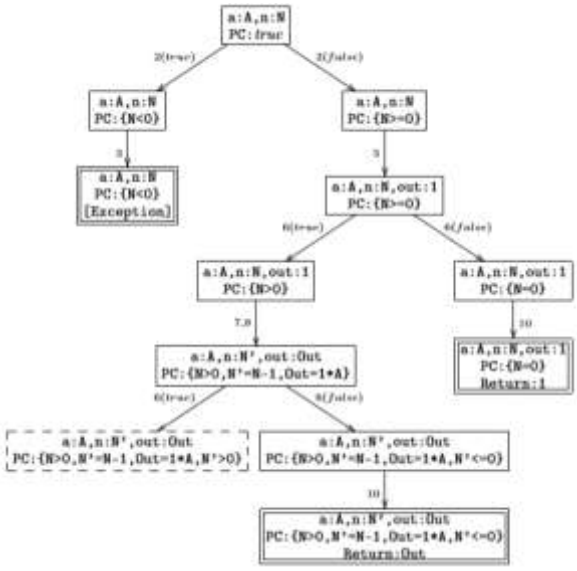

Fig -2. The result of test case generating using symbolic execution

There are two basic techniques of software testing including Black box testing and White box testing [10]. The differences between two these strategies are presented as in below.

Black Box Testing: In Black box testing, the testers do not know about the internal structure, design or implementation of the item being tested. Because of this, the responsibility of

Black box testing is independent to software testers. This technique is applied mainly into high levels of testing, such as: acceptance testing, system testing. Using Black box testing, the implementation and programming knowledge are not required.

The main advantages of Black box testing are [11] reducing the number of test cases; having the independent between programmer and tester and being effective on large units of code. But it also have some disadvantages. For example, it is difficult to design without clear specification. Although there are many symbolic input were generated, a small numbers of possible input can be tested. The repetition of tests is done by programmers.

White Box Testing: In White box testing, the testers do know about the internal structure, design or implementation of the item being tested. The responsibility of White box testing requires software developers. White box testing is often applied into low levels of testing such as: unit testing, integration testing. Both the implementation and programming knowledge are required.

Here are some main advantages and disadvantages of White box testing [11]:

Advantages: Every independent path will be exercised at least once and this is same with all logical decisions. The loops at boundaries were executed. The execution equivalence show the approximation of partitioning.

Disadvantages: The cases omitted in the code were missed out. This needs the skilled tester. This type of testing is nearly impossible to look into every bit of code in order to find out hidden errors.

Apart from these two basic types, gray box testing is known as a combination of Black box and White box testing. The internal structure in gray box testing is partially known.

Based on these strategies of software testing, there are many different techniques that have been used including static testing (manual testing) [13] functional testing, dynamic testing, structural testing and symbolic execution testing [5, 6].

In [2, 12], concepts of constraint programming was introduced. Based on this technique, a dynamic symbolic execution tool for JavaScrip was built. The experimental results showed the effectivity of this proposal. In master thesis, Harris [7] proposed four automata-based symbolic string models for PSE and analyzes their suitability using two criteria: accuracy and performance. The probability computed by these models was compared to the actual probability and the amount of time took to compute it.

In this paper, we intend to introduce a semi-supervised learning method in symbolic execution in order to increase the performance of testing progress. Learning method is applied in the first stage to optimize the libraries or functions of testing process. From this application, the results of testing will be better.

The remaining contents of this paper is organized as follow: Section 2 shows the related researches. Section 3 presents the methodology of proposed model. Experimental results are given in Section 4 and some conclusions are stated in last section.

2. RELATED WORKS

All programs needs to be tested. To do this effectively, people have to know about the fundamental of testing consisting of the principle, goals, techniques and strategies of software testing. In [11], Sawant *et al.* presented in detail every concepts of software testing. The differences among software testing techniques were also explained and the differences between testing and debugging were stated in this paper.

Jamil et al. [8] introduced the existing testing methods based on the mentioned classification of testing method techniques and a software testing life cycle. Besides, the testing process was enhanced by various methods such as automation test, agile, test driven development. In this paper, authors stated that the simulation tool also supported effectively in software testing. Thus, simulation model based software testing techniques would develop.

Symbolic execution is a powerful technique for bug finding and program testing. It is successful in finding bugs in real-world code. The core reasoning techniques use constraint solving, path exploration, and search, which are also the same techniques used in solving combinatorial problems.

Symbolic execution is increasingly used not only in academic settings but also in industry, e.g. in Microsoft, NASA, IBM and Fujitsu, and even at the Pentagon. Many symbolic execution engines have been built targeting different programming languages and architectures. This trend is expected to intensify in the future. Symbolic execution in a distributed setting, leveraging cloud technology, such as Cloud9 [22], SAGE [23], and MergePoint [24], is expected to further extend the applicability of the technique in practice

Symbolic execution used symbol inputs instead of discrete inputs. It was used to check if some certain properties can be violated [3]. The symbolic execution was also integrated with search-based testing in programs with complex heap inputs [4]. In this way, the test cases were generated automatically. Symbolic execution was used in order to generate path conditions. The obtained path conditions were formed as an optimal problem and solved by search-based techniques.

Vanhoef, M., & Piessens [14] showed the way to simulate efficiently cryptographic primitives in symbolic execution progress. This approach was applied successfully in client side implementations of the 4-way handshake of WPA2. A review of symbolic execution and associated tools was

presented by Păsăreanu *et al.* [15]. Moreover, the main challenges of symbolic execution in applying into practical problems were provided. These problems included handling of programs with complex inputs, coping with path explosion and ameliorating the cost of constraint solving. The survey of promising applications and the finding of worst-case execution time in programs, load testing and security analysis, ... was presented.

In symbolic execution, constraint satisfaction problem instances were introduced by Verma and Yap [16]. These instances were used to evaluate the effectiveness of the core techniques in symbolic execution. The benchmarks of this form can spur the development and engineering of improved core reasoning in symbolic execution engines. Solving constraints is one of the most important steps in symbolic execution. Many constraint solvers were proposed in various researches [17-19]. In [19], the transformation of constraints into functions with specific properties was introduced. These functions were called as Satisfaction Functions. The satisfaction function was generated by taking maximum. The values satisfying the corresponding constraint are obtained. The performance this approach was comparable with three constraint solvers that were known to be able to solve non-linear real constraints. Other issues related to constraint solving were also concerned by scientists [20, 21].

3. METHODOLOGY

In this research, we are using the model consisting of following steps:

1. Path bound conditions will be converted to an intermediate format.
2. Solve or simplify the intermediate format.
3. Replace the integer symbol variable with specific conjecture values.
4. Convert intermediate format to automata or bitvector operations.
5. If the binding on automata or bitvector fails, repeat step 3 with other integer specific values in the best set of integer values.
6. The unbound process ends if the matching string value is found or within a limited period of time without finding new integer values, and the unbinding process on the automata and bitvector is invalid.

Automata-based constraint solving reduces the model counting problem to path counting. To count the number of values that satisfy the given constraint within a given domain bound, we count the number of accepting paths in the automaton within the path length bound that corresponds to said domain bound. We use techniques from algebraic graph theory to solve the path counting problem.

This algorithm supplies the procedure that operates on a automaton A passed by reference. This automaton has a track for each variable and term in α. In which:

α is one of the following: a conjunction of numeric and string constraints; a string constraint; a numeric constraint; a string term; or a numeric term.

The operators in this table include:

$\star \in$ {=, ≠, <, ≤, >, ≥, match, ¬ match, contains, ¬ contains, begin, ¬ begin, end, ¬ end }

$\odot \in$ {−, +, ×, length, toin, indexof, lastindexof, reverse, tostring, charat, substring, replacefirst, replacelast, replaceall}.

Then the description of this algorithm is presented as below:

Algorithm-1: Solve the path counting problem.

```
1: if α ≡ α1 ∧ α2 then
2:     SOLVE(A, α1); SOLVE(A, α2);
3:     PROPAGATE(A, α1); PROPAGATE(A, α2);
4: else if α ≡ α1 ⋆ α2 then
5:     SOLVE(A, α1); SOLVE(A, α2);
6:     REFINE(A, ⋆, τ(α1), τ(α2))            ▷ modifies tracks τ(α1) and τ(α2)
7:     PROPAGATE(A, α1); PROPAGATE(A, α2);
8: else if α ≡ ⊙(α1, . . . , αn) then
9:     for all αi ∈ {α1, . . . , αn} do
10:        SOLVE(A, αi);
11:    end for
12:    RESTRICT(A, τ(α), ⊙, τ(α1), . . . , τ(αn));      ▷ modifies track τ(α)
13: end if
```

Many web server requests are compressed and encrypted for efficiency and security before transmission as a network packet. Despite the encryption, a malicious attacker who can observe network packet sizes can use the compression size to learn secret web-session information. Assume an attacker can inject and concatenate his own text with the secret text prior to compression. The smaller the resulting packet, the more compression must have occurred prior to encryption, and so the attacker-controlled input must contain sub-strings which match substrings of the secret text. In the CRIME attack, encryption does not significantly change the size of the packet, as many encryption protocols are size-preserving. Thus, by carefully crafting injected inputs, an attacker can incrementally reveal the secret text.

As just described, we can precisely solve multi-variable linear integer arithmetic constraints by constructing a multi-track binary integer automaton that recognizes tuples of solutions. However, integer variable solutions can be related to string variables through operations that have both string and integer parameters such as length or index of. Given the Deterministic Finite Automaton (DFA) representing the solutions for integer variables, we must propagate the constraints imposed by the integer solutions to each related string variable. We do so by first converting the binary DFA solution representation A for an integer variable i to a set comprehension representation S.

We described how to propagate solutions from binary integer DFA to string DFA. In order to propagate solutions from string DFA to binary integer DFA, we reverse this

process by converting a string DFA into a unary length DFA, extracting the semi-linear set, and constructing the corresponding binary integer DFA.

By using the notation as below:

$\odot \in$ {–, +, ×, length, toin, indexof, lastindexof, reverse, tostring, charat, substring, replacefirst, replacelast, replaceall}.

This algorithm also provides the procedure that operates on a automaton *A* passed by reference. This automaton has a track for each variable and term in α. The detail of the algorithm is introduced as follow:

Algorithm -2: propagate solutions from binary integer DFA to string DFA.

```
1: if φ ≡ ⊙(α_1, ..., α_n) then
2:     REFINE(A, τ(α), ⊙, τ(α_1), ..., τ(α_n));   ▷ modifies tracks τ(α_1) to τ(α_n)
3:     for all α_i ∈ {α_1, ..., α_n} do
4:         PROPAGATE(A, α_i);
5:     end for
6: else if φ ≡ φ_1 ∧ φ_2 then
7:     PROPAGATE(A, φ_1); PROPAGATE(A, φ_2);
8: else if φ ≡ φ_1 ∨ φ_2 then
9:     A_φ1 = A ∩ A_φ1; A_φ2 = A ∩ A_φ2;
10:    PROPAGATE(A_φ1, φ_1); PROPAGATE(A_φ2, φ_2);
11: end if
```

Since the negation operator is non-monotonic and since we sometimes over-approximate the solution sets of subformulas, before the automata construction, we convert the input formula to negation normal form by pushing negations to atomic formulas. For disjunctions, each disjunct has its own automaton. Then, the automaton for the disjunction corresponds to the automaton that accepts the union of sets accepted by each disjunct automaton. We compute the union automaton using automata product.

4. CONCLUSIONS

Constraint solvers are well-known tools for solving many real-world problems such as theorem proving and real-time scheduling. One of the domains that strongly relies on constraint solvers is the technique of symbolic execution for automatic test data generation. Many researchers have tried to alleviate the shortcomings of the available constraint solvers to improve their applications in symbolic execution for test data generation. Despite many recent improvements, constraint solvers are still unable to efficiently deal with certain types of constraints. In particular, constraints that include non-linear real arithmetic are among the most challenging ones.

In this research, we introduce two algorithms that are used in solving the constraints and propagating solutions from binary integer DFA to string DFA.

Based on this, we can find new integer constraints incorporated with string constraints.

In future studies, we will try to apply these approaches to an application. Then, the effectivity of proposed algorithms is evaluated.

REFERENCES

[1]. Albert, E., Arenas, P., Gómez-Zamalloa, M., & Rojas, J. M. (2014, June). Test case generation by symbolic execution: basic concepts, a CLP-based instance, and actor-based concurrency. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems* (pp. 263-309). Springer, Cham.

[2]. Amadini, R., Andrlon, M., Gange, G., Schachte, P., Søndergaard, H., & Stuckey, P. J. (2019, June). Constraint Programming for Dynamic Symbolic Execution of JavaScript. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research* (pp. 1-19). Springer, Cham.

[3]. Baldoni, R., Coppa, E., D'elia, D. C., Demetrescu, C., & Finocchi, I. (2018). A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, *51*(3), 50.

[4]. Braione, P., Denaro, G., Mattavelli, A., & Pezzè, M. (2017, July). Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 90-101). ACM.

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

[6]. Gaston, C., Le Gall, P., Rapin, N., & Touil, A. (2006, May). Symbolic execution techniques for test purpose definition. In *IFIP International Conference on Testing of Communicating Systems* (pp. 1-18). Springer, Berlin, Heidelberg.

[7]. Harris, Andrew, "Suitability of Finite State Automata to Model String Constraints in Probablistic Symbolic Execution" (2019). *Boise State University Theses and Dissertations*. 1597.

[8]. Jamil, M. A., Arif, M., Abubakar, N. S. A., & Ahmad, A. (2016, November). Software Testing Techniques: A Literature Review. In *Information and Communication Technology for The Muslim World (ICT4M), 2016 6th International Conference on* (pp. 177-182). IEEE.

[9]. King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, *19*(7), 385-394.

[10]. Nidhra, S., & Dondeti, J. (2012). Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, *2*(2), 29-50.

[11]. 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.

[12]. Søndergaard, H., & Stuckey, P. J. (2019, June). Constraint Programming for Dynamic Symbolic Execution of JavaScript. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4–7, 2019, Proceedings* (Vol. 11494, p. 1). Springer.
[13]. Shyam-Sunder, L., & Myers, S. C. (1999). Testing static tradeoff against pecking order models of capital structure1. *Journal of financial economics*, *51*(2), 219-244.
[14]. Vanhoef, M., & Piessens, F. (2018). Symbolic execution of security protocol implementations: handling cryptographic primitives. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*.
[15]. Păsăreanu, C. S., Kersten, R., Luckow, K., & Phan, Q. S. (2019). Symbolic Execution and Recent Applications to Worst-Case Execution, Load Testing, and Security Analysis. In *Advances in Computers* (Vol. 113, pp. 289-314). Elsevier.
[16]. Verma, S., & Yap, R. H. (2019, November). Benchmarking Symbolic Execution Using Constraint Problems-Initial Results. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)* (pp. 1-9). IEEE.
[17]. Amadini, R., Andrlon, M., Gange, G., Schachte, P., Søndergaard, H., & Stuckey, P. J. (2019, June). Constraint Programming for Dynamic Symbolic Execution of JavaScript. In International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (pp. 1-19). Springer, Cham.
[18]. Kapus, T., Nowack, M., & Cadar, C. (2019, October). Constraints in Dynamic Symbolic Execution: Bitvectors or Integers?. In International
[19]. Amiri-Chimeh, S., & Haghighi, H. (2019). An approach to solving non-linear real constraints for symbolic execution. Journal of Systems and Software, 157, 110383.
[20]. Chekam, T. T., Papadakis, M., Cordy, M., & Traon, Y. L. (2020). Killing Stubborn Mutants with Symbolic Execution. arXiv preprint arXiv:2001.02941.
[21]. Mukherjee, R., Joshi, S., O'Leary, J., Kroening, D., & Melham, T. (2020). Hardware/Software Co-verification Using Path-based Symbolic Execution. arXiv preprint arXiv:2001.01324.
[22]. L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A Software Testing Service. SIGOPS Oper. Syst. Rev., 43(4):5–10, Jan. 2010
[23]. P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. Queue, 10(1):20:20–20:27, Jan. 2012.
[24].] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing Symbolic Execution with Veritesting. In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM.

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.

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. Her major interests are discrete mathematics, Monte Carlo method, optimization, probability theory and statistics, machine learning.