

A hybrid constraint solver combined with meta-heuristic search

Xiaomei Hou^{1,3}, Mengxiang Lin^{1,3} *, Tahir Jameel^{1,2}, and Dongmei Chen⁴

¹ State Key Laboratory of Software Development Environment, Beihang University, China

² School of Computer Science and Engineering, Beihang University, China

³ School of Mechanical Engineering and Automation, Beihang University, China

⁴ Department of Mechanical Engineering, University of Texas at Austin, U.S.A

Abstract. Constraint solving is one of the essential parts of symbolic execution. Most of the constraint solvers are aimed at a particular kind of constraint and might have no way to deal with complex path constraints derived from real-world applications. To tackle the problem, we propose a hybrid solving strategy for path constraints containing multiple kinds of constraints. In particular, a split approach is used to cut the whole path constraint into two parts that can be solved by different constraint solving techniques. a local search method of meta-heuristics is combined with an off-the-shelf constraint solver to solve complex mathematical constraints. We have extended the constraint solver in the symbolic execution engine KLEE with our approach and evaluated its effectiveness on a set of mathematical programs. The results show that our approach can solve a majority of complex path constraints in reasonable time.

1 Introduction

The success of dynamic symbolic execution [2, 3] is driving a newfound interest in this traditional technique. As a result, symbolic execution has been applied to many fields of software analysis and testing in a variety of ways. However, symbolic execution does have its limitations, especially in constraint solving. The constraint systems generated by symbolic execution contain mixed data types including integers, strings, floating-points, arrays and records and so on and can be much complex. Off-the-shelf constraint solvers specific to a particular kind of constraint might have no way to deal with complex path constraints derived from real-world applications.

In our prior work [5], we have proposed a lazy evaluation approach for the program structures in modern programming languages that cannot be executed symbolically. To solve a path constraint with intermediate variables introduced by lazy symbolic evaluation, we split the path constraint into two parts and solve them separately. In this paper we further propose a hybrid constraint solving approach for path constraints containing multiple kinds of constraints.

* corresponding author. This work is supported by China Scholarship Council.

Specifically, a local search method of meta-heuristics is combined with an off-the-shelf constraint solver to solve complex mathematical constraints. The main contribution of this paper is as follows: (1) to propose the approach of integration of local search into off-the-shelf constraint solvers; (2) to extend the constraint solver in the symbolic execution engine KLEE [3] with our approach; and (3) to evaluate the effectiveness of our approach on a set of mathematical programs.

In the rest of this paper, we give an overview of our approach with an example in Section 2. The details of our approach are presented in Section 3 and our experimental setup and results are described in Section 4. Section 5 discusses the related work and Section 6 is a summary of the paper.

2 Example

We use the simple program shown in Figure 1 to illustrate how our method works. Let x_s and y_s denote the symbolic values of the input variables x and y , respectively. For the target path $T = \langle 1, 2, 4, 5, 6, 8, 9, 10 \rangle$, the corresponding path constraint PC is $(x_s - y_s \leq 10) \wedge (x_s^2 + y_s^2 + x_s * y_s \geq 200) \wedge (x_s + y_s < 20) \wedge (\exp(x_s * y_s) > 400) \wedge (y_s \geq 0)$. The feasible region of PC is shown in Figure 2. It is well known that the constraints like PC are very expensive to deterministically solve.

```

1 void test(float x, float y){
2   if(x - y > 10)
3     S0;
4   else if(x*x + y*y + x*y >= 200)
5     S1;
6   if(x + y >= 20)
7     S2;
8   else if(exp(x*y) > 400 && y >= 0)
9     S3;
10 }
```

Fig. 1. Example in C-like syntax.

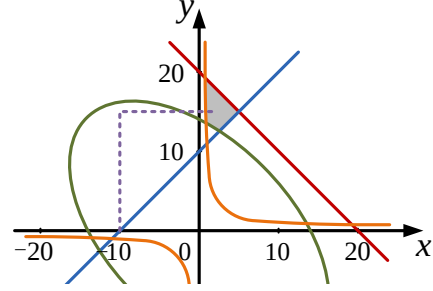


Fig. 2. Feasible region of PC

As in our prior work [5], we continue to distinguish constraints according to how difficult it is to solve them. For our example, $(x_s - y_s \leq 10) \wedge (x_s + y_s < 20) \wedge (y_s \geq 0)$ in PC is linear, denoted as PC_{linear} , while the rest is non-linear, denoted as $PC_{non-linear}$. We first pass PC_{linear} into a linear constraint solver. A test input that exercises the path T is generated if the solution returned happens to satisfy $PC_{non-linear}$. Beyond that, an input value might be derived by simplifying and solving $PC_{non-linear}$ using the solution to PC_{linear} [9]. However, it is very unlikely in a situation like this even though we strive to solve PC_{linear} again and again. To address the issue in a general way, we combine meta-heuristic search to solve $PC_{non-linear}$. Instead of assigning initial random values, we confine the initial inputs to the solutions to PC_{linear} . The solutions

to a part of PC provide a good start for search and may be able to help find a solution faster than the random values in the whole search space. The dashed line in Figure 2 presents a possible trace of search by using local search.

3 Approach

Different techniques suited for different kinds of constraints have been developed. We argue that diverse techniques must be brought together to solve such constraint systems involving multiple kinds of constraints like path constraints. In particular, a hybrid solving approach combining local search is proposed in this paper. Our approach proceeds in three steps from a high level. First, the path constraint derived by symbolic execution PC is transformed into the disjunctive normal form (DNF), i.e. $PC = C_1 \vee \dots \vee C_i \vee \dots \vee C_n$, where C_i is defined as a satisfied formula of PC . The satisfied formula is denoted by $C = p_1 \wedge \dots \wedge p_i \wedge \dots \wedge p_m$, where p_i is a primitive constraint, i.e. the relational expression in a branch condition. For details on this process refer to [5]. Second, we split the satisfied formula C and pass the linear part of it into a linear constraint solver. Third, a local search utilizing the solutions to the linear part is performed to find a feasible solution to PC .

Our local search method is a variation of the alternating variable method (AVM) [1]. The search procedure is initialized with the solution to the linear part instead of random values. The fitness function defined is: $F = \sum_{1 \leq i \leq m} \omega_i * d_i$, where ω_i is a weight associated to each primitive constraint and d_i is a distance function as shown in Table 1. Similar to [10], we introduce the notion of weight to guide the search for unsatisfied primitive constraints. The weights are reduced to zero if the corresponding primitive constraints are satisfied, or they are increased after the number of count times specified is reached.

Table 1. Constraint operators and corresponding distance function

Operator	Distance function
$a == b$	$ a - b $
$a != b$	if $a - b == 0$ then 1 else $- a - b $
$a < b$	$a - b$
$a \leq b$	$a - b$
$a > b$	$b - a$
$a \geq b$	$b - a$

AVM unceasingly modifies the values of one input variable until it does not find a better fitness. This process is referred to as the iteration. To speed up the iterative process, some optimization strategies developed in recent work [2, 9, 11] are considered. For example, we try to simplify the satisfied formula using the concrete values of input variables explored at the end of the iteration. The linear constraint solver is invoked if the simplified constraint is linear. Otherwise, a new

input variable is randomly selected from all of unsatisfied primitive constraints and the next iteration starts. A search process ends when all of the input variables have been explored or a solution is found. In the former case, a restart is needed to make a new search.

Restart is a simple but effective way to overcome local optima. As before, we confine the initial values to the solutions to the linear part of a satisfied formula. However, extra constraints are added to the linear part in order to explore diverse space. For example, suppose that (S_x, S_y) is the position where the search is stuck, the domain of input variable x is $[L_x, R_x]$, and the domain of input variable y is $[L_y, R_y]$. The search space would be therefore subdivided into four parts by (S_x, S_y) . Furthermore, an extra constraint specified to a partition is selected at random, for example, $L_x \leq x < S_x$ and $S_y < y \leq R_y$.

4 Evaluation

KLEE [3] is a well-known symbolic execution engine, in which the SMT solver STP [8] for bit-vectors and arrays is used by default. We extended the constraint solver in KLEE with our approach. In order to support floating point types and their basic operations, floating point related functions in KLEE-FP [4] are introduced in our implementation Hybrid-KLEE. Furthermore, we have evaluated Hybrid-KLEE on seven complex mathematical programs shown in Table 2. Five out of the seven programs are taken from GNU Scientific Library (GSL) [7] and the remaining two programs are chosen from the book [14]. All of our experiments are run on a 4 core processor VMware virtual machine with 2 GB memory running Ubuntu 10.4.

Table 2. Complex mathematical programs in the study

Program	Description
golden	an implementation of goldensection minimum finding algorithm
solve_cubic	find the real roots of $x^3 + ax^2 + bx + c = 0$
solve_quadratic	find the real roots of $ax^2 + bx + c = 0$
zsolve_cubic	find the complex roots of $x^3 + ax^2 + bx + c = 0$
zsolve_quadratic	find the complex roots of $ax^2 + bx + c = 0$
Cei	calculate the standard exponential integral $E_i(x) \equiv \int_{-\infty}^x \frac{e^t}{t} dt, x < 0$
CbetaBN	calculate the normalized incomplete beta function $B_N(x, a, b) \equiv \frac{B_I(x, a, b)}{B(a, b)} \quad a, b > 0, 0 \leq x \leq 1$

Table 3 details the results of our experiments. The column LOC and INS list the number of lines of source code and LLVM instructions executed, respectively. The number of paths explored is shown in the column Path while the number of tests generated is shown in the column Test. The column ICOV presents the percentage of LLVM instructions covered and BCOV is the percentage of branches covered. The last two columns in Table 4 give the total execution time and the percentage of time spent in constraint solving respectively.

Table 3. Results of our experiments

PRO	LOC	INS	Path	Test	ICOV(%)	BCOV(%)	Time(s)	Solving(%)
gupta	39	234	8	8	100.00	100.00	307.84	99.97
gsl_min_golden	78	1155	13	13	86.42	70.00	1.17	98.27
gsl_poly_solve_c	82	371	6	4	83.33	70.00	0.66	99.09
gsl_poly_zsolve_c	126	550	6	4	82.31	72.73	0.94	90.14
gsl_poly_solve_q	55	282	9	9	100.00	100.00	0.45	55.19
gsl_poly_zsolve_q	78	394	9	9	100.00	100.00	1.32	59.99
Cei	58	10494	21	21	93.71	83.33	576.16	99.87
CbetaBN	86	1903	7	7	96.51	70.00	2.29	97.05

Of all our experiments, Hybrid-KLEE achieves more than 82.31% line coverage and 70% or over branch coverage. Two programs get 100% line coverage and branch coverage. Furthermore, we have scrutinized the branches uncovered by Hybrid-KLEE. Some branches failed to be covered are related to the poor performance of local search on such constraints like non-linear equality. Others are connected with the loops depending on inputs. A couple more iterations are required during symbolic execution in order to cover these branches.

5 Related works

Instead of using an off-the-shelf constraint solver in a black box way, the recently developed constraint solving techniques consider more of the characteristics of path constraints produced by symbolic execution. Here, we focus on the most closely related works to ours. Similar to our prior work [5], [9] delays (or lazy) concretization of unknown or complex functions during symbolic execution and splits the resulting path constraint into simplePC and complexPC. The solutions obtained by simplePC are used to compute and to simplify complexPC. However, multiple solutions for simplePC might not help much with complex constraints such as those in our experiments. To address the issue, we further develop a method combined with meta-heuristics in this paper. FloPSy [6] extends the custom arithmetic solvers in Pex with the alternating variable method and genetic algorithms to tackle the problem of floating point computations in DSE. The search process operates in relative isolation and performs a random restart to overcome local optima in FloPSy. In contrast, our search process starts with the solutions produced by the solvable constraints in path constraint. Several constraint solvers based on meta-heuristic search [10, 12, 13] are proposed recently in order to handle complex mathematical constraints. However, our hybrid solver focuses on the integration of meta-heuristic search into off-the-shelf constraint solvers.

6 Conclusion

A richer set of data types and operations makes path constraints extremely difficult to solve. We have developed and implemented the hybrid approach that

combine local search with off-the-shelf constraint solvers to solve complex mathematical constraints. Preliminary experimental results show the effectiveness of our approach. Introduction of other constraint solving techniques and more experimental comparison with other approaches are left as future work.

References

1. Korel, B.: Automated Software Test Data Generation. *Software Engineering, IEEE Transactions on* 16, no. 8 (1990): 870-879.
2. Patrice Godefroid, Nils Klarlund and Koushik Sen: DART: Directed Automated Random Testing. In: *Proceedings of the 26th Programming Language Design and Implementation*, pp.213-223. ACM (2005).
3. Cristian Cadar, Daniel Dunbar, Dawson Engler: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pp.209-224. USENIX Association (2008).
4. Peter Collingbourne, Cristian Cadar and Paul H. J. Kelly: Symbolic Crosschecking of Floating-Point and SIMD Code. In: *Proceedings of the 6th ACM European Conference on Computer Systems*, pp.315-328. ACM (2011).
5. Lin Meng-xiang, Chen Yin-li, Yu Kai and Wu Guo-shi: Lazy Symbolic Evaluation and its Path Constraints Solution. In: *Automation of Software Test(AST), ICSE Workshop on*, pp.79-87. IEEE(2009).
6. Kiran Lakhotia, Nikolai Tillmann, Mark Harman and Jonathan de Halleux: FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution. In: *Proceedings of the 25th IFIP International Conference on Testing Software and Systems*, pp.142-157. Springer(2010).
7. GNU Scientific Library, <http://www.gnu.org/software/gsl/>
8. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: *Proceedings of the 19th International Conference on Computer Aided Verification*, pp.524-536. Springer (2007)
9. Corina S. Păsăreanu, Neha Rungta and Willem Visser: Symbolic Execution with Mixed Concrete-Symbolic Solving. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis(ISSTA)*, pp.34-44. ACM (2011).
10. Mateus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Păsăreanu: CORAL: Solving Complex Constraints for Symbolic PathFinder. In: *Proceedings of the NASA Formal Methods*, pp.359-374. Springer (2011)
11. Ikpe Erete, and Alessandro Orso: Optimizing constraint solving to better support symbolic execution. In: *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pp.310-315. IEEE(2011).
12. Mateus Borges, Marcelo d'Amorim, Saswat Anand, David Bushnell, Corina S. Păsăreanu: Symbolic Execution with Interval Solving and Meta-heuristic Search. In: *Software Testing, Verification and Validation (ICST), IEEE Fifth International Conference on*, pp.111-120. IEEE(2012).
13. Roberto Bagnara, Matthieu Carlier, Roberta Gori, Arnaud Gotlieb: Symbolic Path-Oriented Test Data Generation for Floating-Point Programs. In: *Software Testing, Verification and Validation (ICST), IEEE Sixth International Conference on*, pp.1-10. IEEE(2013).
14. Yang Hua-zhong, Wang Hui: Numerical computing method and C language function library. Science Press, Beijing (1996)