8-2019

InSPeCT: Iterated local search for solving path conditions

Fuxiang CHEN
Hong Kong University of Science and Technology

Aldy GUNAWAN
Singapore Management University, aldygunawan@smu.edu.sg

David LO
Singapore Management University, davidlo@smu.edu.sg

Sunghun KIM
Hong Kong University of Science and Technology

Published in 2019 15th IEEE International Conference on Automation Science and Engineering (CASE): Vancouver, August 22-26, pp. 1724-1729.
DOI: 10.1109/COASE.2019.8843039

InSPeCT: Iterated Local Search for Solving Path Conditions

Fuxiang Chen[1,3], Aldy Gunawan[2], David Lo[2] and Sunghun Kim[1]

***Abstract*— Automated test case generation is attractive as it can reduce developer workload. To generate test cases, many Symbolic Execution approaches first produce Path Conditions (PCs), a set of constraints, and pass them to a Satisfiability Modulo Theories (SMT) solver. Despite numerous prior studies, automated test case generation by Symbolic Execution is still slow, partly due to SMT solvers' high computationally complexity. We introduce InSPeCT, a Path Condition solver, that leverages elements of ILS (Iterated Local Search) and Tabu List. ILS is not computational intensive and focuses on generating solutions in search spaces while Tabu List prevents the use of previously generated infeasible solutions. InSPeCT is evaluated against two state-of-the-art solvers, MLB and Z3, on ten Java subject programs of varying size and complexity. The results show that InSPeCT is able to solve 16% more PCs than MLB and 41% more PCs than Z3. On average, it is 103 and 5 times faster than Z3 and MLB, respectively. It also generates tests with higher test coverage than both MLB and Z3.**

I. Introduction

Testing is an essential component in ensuring the quality of deliverable software. However, it is both an expensive and a time consuming activity [1]. Thus, automated test case generation is attractive as it can help reduce developers' workload. To generate tests with high coverage, various Symbolic [2] and Concolic [3] testing techniques have been introduced [4] [5]. These approaches first use Symbolic Execution to produce Path Conditions (PCs) – a set of constraints consisting of unsolved variables, arithmetic and relational operators – and then pass them to a Satisfiability Modulo Theories (SMT) solver such as Z3 [6]. The solver then analyzes the PCs and produces a solution (a mapping of variables to values that solves the constraints) for the solvable PCs. Each mapping of solved variable values guides the execution flow of the code and serves as a test case. Despite numerous prior studies, automated test case generation is still slow, partly due to the high computational cost of SMT solvers. Some constraints, especially non-linear arithmetic constraints, are challenging.

Over the last decade, application of (meta)heuristics, such as Simulated Annealing [7] and Tabu Search [8], in Software Engineering (*aka.* Search-Based Software Engineering) has risen to become an active research area [9] [10].

[1]Fuxiang Chen and Sunghun Kim are affliated with Clova AI Research, Naver Corp. and with the Department of Computer Science and Engineering, the Hong Kong University Science and Technology fchenaa@cse.ust.hk , hunkim@cse.ust.hk

[3]Fuxiang Chen is with DeepSearch, Inc. cfuxiang@deepsearch.com

[2]Aldy Gunawan and David Lo are with the School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore aldygunawan@smu.edu.sg, davidlo@smu.edu.sg

We propose InSPeCT (**I**terated Local **S**earch for **P**ath **C**ondi**T**ion), a novel technique leveraging Iterated Local Search (ILS) and Tabu Lists for solving PCs. ILS focuses on generating solutions in search spaces, while using a Tabu List prevents the use of previously generated infeasible solutions. InSPeCT is composed of two main phases: the Randomization Phase, and the Improvement Phase. The Randomization Phase uses simple heuristics to solve PCs while the Improvement Phase focuses on solving the remaining unsolved PCs by generating more possible input values.

InSPeCT is able to solve a total of 1,104 (88.5%) PCs from a standard benchmark suite consisting of ten Java subject programs and it is consistent in its solvability performance with small variance (out of five runs, the number of PCs solved has a minimum variance of 0.56 and a maximum variance of 4.56 out of the ten subjects). Each Phase/Strategy is also able to solve a reasonable number of PCs that were previously unsolvable in the earlier Phase/Strategy. InSPeCT, when compared to existing state-of-the-art SMT solvers, MLB [11] and Z3 [6], solves 16% more PCs than MLB and 41% more PCs than Z3. InSPeCT also achieves higher test coverage as compared to MLB and Z3. Furthermore, InSPeCT is more efficient than MLB and Z3. It is 103× faster than Z3 and 5× faster than MLB, on average.

II. Related Work

Metaheuristics, such as Simulated Annealing [13] and Tabu Search [8], have been widely applied in solving combinatorial optimization problems. However, the application of metaheuristics in software engineering has only began to gain attention within the last decade [9] [14]. Clarke et al. [14] summarize the application of metaheuristic algorithms for solving problems in software engineering and report that the most widely used metaheuristic in software testing is Genetic Algorithm [12], [15], followed by Simulated Annealing [13] [7].

Diaz et al. [16] introduce TSGen, an automatic generator of tests for a given program. This is reported to be the first work incorporating Tabu Search in order to perform automatic test generation. They introduce two cost functions to intensify and diversify the search, and a process to find neighborhoods. Sahin and Akay [17] compare several metaheuristic algorithms on software test data generation. The Artificial Bee Colony [18], Particle Swarm Optimization [19], Differential Evolution [20] and Firefly Algorithms [21] are used for problems that need to find maximum values.

In Software Engineering, particularly in the field of Symbolic Execution, PCs are generally solved by a SMT solver

such as Z3 [6]. Z3, developed by Microsoft, has been reported to solve more PCs than other SMT solvers such as CVC3 [22] and Yices [23]. Z3 incorporates multiple theories to reason about and generate solutions for PCs. Z3 has been reported to have difficulty in solving complex PCs such as those involving non-linear operations [24]. Dinger and Agha [25] introduce the Concolic Walk (CW) algorithm, combining linear constraint solving and heuristic search to solve complex arithmetic path conditions. CW differentiates between linear and non-linear constraints, and uses an off-the-shelf solver to solve linear constraints. However, CW is only evaluated on a corpus of small programs of up to 335 LOC.

Recently, Li et al. [11] proposed a new symbolic execution tool, MLB, that is based on Machine Learning Based constraint solving to solve path conditions. MLB transforms the path conditions into optimization problems. It adapts RACOS [26], a Machine Learning based optimization technique, by using the dissatisfaction degree as the objective function. It is reported that MLB outperforms CW [25], SPF-CORAL [27], SPF-Mixed [28] and jCUTE [29] in terms of the number of solved PCs.

III. Background

Symbolic Execution (SymEx) is a technique for analyzing source code to determine what inputs cause which part of the code to be executed. SymEx is commonly used in Software Testing to identify test cases (software inputs) that cause different parts of the software to be executed. SymEx executes the code with symbols as arguments and at each decision point (e.g. the test of an *if* statement), it collects the conditions leading to the choice that is made. Such a sequence of conditions is known as a *Path Condition* (PC).

A PC resembles a logical formula with variables and different arithmetic operators (addition, subtraction, multiplication, division and modulo), relational operators (>=, <=, >, <, == and $\neq$) and logical operators (*AND* and *OR*). A PC may consist of multiple subconstraints refered to as branches. Solving a PC requires providing actual values to the variables that will cover the respective path. A PC can be linear or non-linear where the latter is harder to solve, and can involve variables of various data types (e.g. *Integer*, *Floating point*, *Strings*, etc). Solving constraints involving variables of *Integer* data types is known to be a hard problem. In our paper, we are focusing on PCs that are non-linear and consist of integer-typed variables.

Fig. 1 shows an example of a piece of code (lines 1 - 8) and one possible PC. The code involves three variables *i0*, *i1* and *i2*. When this PC is passed to a SMT solver, the solver returns a possible mapping of variables to values that satisfies the constraints, as shown in line 11. This variable mapping represents a test case that can drive the code execution from the beginning to line 8.

Solving more PCs implies that more test cases will be generated to cover more paths. Thus, more code will be typically covered (tested) when more PCs are solved. SMT solvers are usually used in Software Testing to solve PCs

```
1  int i0 = readInput ;
2  int i1 = readInput ;
3  int i2 = readInput ;
4
5  if (i0 > 100)
6    if (i1 >= 2)
7      if (i2 > i1)
8        i1 = i1 * i2 ;
8  halt
9
10 PC: (i0 > 100) && (i1 >= 2) && (i2 > i1)
11 Solved: i0 = 101, i1 = 2, i2 = 3
```

Fig. 1. An example code listing and a PC with its solved values

automatically. Fundamentally, SMT solvers depend on SAT solvers to determine the satisfiability of the PC.

Talbi et. al [30] define a metaheuristic as a search methodology or an algorithm for finding a near optimal solution to an optimization problem within a solution space. We apply Iterated Local Search (ILS) [31], a procedure to randomly generate solutions and includes a perturbation mechanism that makes it possible to escape local optima. In order to avoid unnecessary analysis, we also include a feature of Tabu Search, namely the Tabu List [8], which is responsible for keeping track of all solutions (e.g. the variable values that could not solve some PCs) that have been tested in previous iterations. Our problem deals with discrete values of integer-typed variables and our approach does not have an explicit objective function, which is commonly used in discrete optimization problems. However, we still have an objective, which is to find a set of input variables that satisfies a set of PCs.

IV. Approach

InSPeCT (**I**terated Local **S**earch for **P**ath **C**ondi**T**ion), consists of two main phases, namely (1) **Randomization** and (2) **Improvement**. In the Randomization phase, a simple heuristic for quickly solving some PCs is introduced by generating variable values randomly. We then check whether the resulting set of variable values can solve any PCs. If the set of selected variable values is able to solve some PCs, those solved PCs are removed from further consideration.

In the Improvement phase, the remaining PCs are solved by using ILS to guide the selection of variable values. The idea is to randomly generate values. However, in some cases, InSPeCT can be trapped at certain values and is unable to locate other values. Therefore, a perturbation strategy is applied to further explore other possible values. We iteratively adjust the variable values by adding or subtracting some values to/from them.

A. Randomization Phase

This phase follows a simple heuristic for generating an initial set of variable values within lower and upper bound values for solving PCs. PCs that are solved according to these values are considered solvable. Let P be a set of PCs that needs to be solved. $P = P^* \cup P'$ where P^* and P' represent the set of solved and unsolved PCs, respectively.

Algorithm 1 RANDOMIZATION (PC)

1: $P^* \leftarrow \emptyset$
2: $P' \leftarrow P$
3: $i = 0$
4: **while** the stopping criteria is not met **do**
5: **for all** $j = 1$ to $|N|$ **do**
6: $i \leftarrow i + j$
7: Set $[LB_n, UB_n]$ = $[-i, i]$ $(\forall n \in N)$
8: Generate a variable value for each variable n within $[LB_n, UB_n]$ $(\forall n \in N)$
9: **while** all PCs in P' have not been tested **do**
10: Select one PC randomly from P'
11: Check whether the set of generated values is in the Tabu List
12: **if** the set is not in the Tabu List **then**
13: Solve the selected PC with generated variable values
14: **if** the selected PC is solvable **then**
15: $P^* \leftarrow P^* \cup$ {the selected PC}
16: $P' \leftarrow P' \setminus$ {the selected PC}
17: **else**
18: Store the set of generated $|N|$ variable values in the Tabu List
19: **end if**
20: **end if**
21: **end while**
22: **end for**
23: **end while**
24: **return** P^* and P'

Algorithm 2 IMPROVEMENT (P, P', P^*)

1: Set $[LB_n, UB_n]$ = [-1,1] $(\forall n \in N)$
2: **while** all PCs in P' have not been tested **do**
3: Select one PC randomly from P'
4: Generate a set of n input variables with values, within $[LB_n, UB_n]$ $(\forall n \in N)$ and ensure that it is not in the Tabu List of the selected PC
5: Solve the selected PC with the generated input variables
6: **if** the selected PC is solvable **then**
7: $P^* \leftarrow P^* \cup$ {the selected PC}
8: $P' \leftarrow P' \setminus$ {the selected PC}
9: **else**
10: **for all** $k = 1$ to 2 **do**
11: Store a set of generated n input variables in the Tabu List of the selected PC
12: Apply STRATEGY k
13: **if** the selected PC is solvable **then**
14: $P^* \leftarrow P^* \cup$ {the selected PC}
15: $P' \leftarrow P' \setminus$ {the selected PC}
16: **break**
17: **end if**
18: **end for**
19: **end if**
20: **end while**
21: **return** P^* and P'

At first, $P^* = \emptyset$ and $P' = P$ (lines 1-2). A Tabu List is used to keep track of the mappings that have been generated, but were unable to solve that PC.

Let N be the set of variables appearing in the PCs. We define lower and upper bound discrete values, LB_n and UB_n for each variable n, which are initialized to -1 and 1, respectively (lines 5-7). A random number between the lower and upper bound values (inclusive) is generated for each variable independently (line 8). If we are unable to find values that can solve the PC, the lower and upper bound values are increased by a particular value iteratively, up to $|N|$, as a threshold.

A PC from P' is selected randomly (line 10). The set of generated variable values is checked to ensure that it is not contained in the selected PC's Tabu List (line 11). This set of randomized values is then used by a scripting engine to determine whether they solve the PC. If the set of randomized values solves a PC, the PC is added to P^* (line 15) and removed from P' (line 16), indicating that the PC has been solved. Otherwise, the set of randomized values is added to the PC's Tabu List (line 18). This current set of randomized values is then used to evaluate other PCs until all PCs in P' are tested (lines 9-21).

We then increment LB_n and UB_n (lines 5-6) and then repeat the process. This process continues until a certain time limit has passed (*stopping criteria*) (line 4). At the end, the algorithm returns two sets, P^* and P' (line 24).

B. Improvement Phase

We apply ILS to solve the remaining PCs by generating more possible input values through some strategies. First, we describe how other set of possible variable values are selected by adjusting the lower and upper bound values of the relevant variables. In order to further improve the chance of solving P', we propose two different strategies to adjust the lower and upper bound values of variables. Both are applied one after the other. The algorithm is outlined in Algorithm 2.

We initially set the lower and upper bound values for all variables to be -1 and 1 respectively (line 1). For each PC that has not been solved, a random value for each variable found in that PC is chosen. This set of randomized values is checked against the Tabu List of that PC. If the set exists, another set of values will be generated, until the new set is not in the PC's Tabu List (line 4). If the new set of randomized values can solve the PC (line 6), $P*$ and P' are updated accordingly (lines 7-8) and proceed to the next PC (line 2). Otherwise, the two strategies, STRATEGY 1 AND STRATEGY 2, are applied (lines 10-18). If a strategy is able to solve the PC, we update $P*$ and P' accordingly (lines 14-15) and terminate the process. Otherwise, we continue to the next strategy. This would be stopped if either one of the strategies is able to solve the PC or we have reached the last strategy (lines 10-18).

1) Strategy 1: All variables are set to their initial value, as selected in line 4 of Algorithm 2, and then each variable is selected one at a time based on the frequency of its appearance (in descending order) in the selected PC. The value of the variable with the highest frequency is first incremented by 1. The new set of mapping values is then evaluated to see if it can solve the PC. This process continues for i iterations. Our experimental study showed that 10 is the best threshold for i.

After going through all the variables, one at a time, if the PC is still unsolvable, the algorithm will reset all the variables to the initial values (Algorithm 2 Line 4) and

increment all the variables together by 1 for the same i iterations. The algorithm again checks on each iteration if the PC is solvable according to the resulting values. If the PC is still not solved at the end of i iterations, the algorithm resets all the variable values to the initial values (line 4 of Algorithm 2) and repeats the same process by increasing incrementally by a big delta value, instead of 1. We experimentally found that 100 is the best threshold for the big delta value. For example, if the initial variable value is 1, then the new variable value will be 101 after increasing by 100. Again, if the PC is still unsolvable after increasing all the variables independently, all the variables are then incremented at once by the same big delta value for the same i iterations. If the PC is still unsolved, it will flow into Strategy 2.

2) Strategy 2: If the PC is still not solved with the new sets of values, InSPeCT proceeds to STRATEGY 2. STRATEGY 2 is similar to STRATEGY 1 except that it decreases the variable values starting with the original mapping obtained in Algorithm 2 line 4, instead of increasing them. If the PC still could not be solved with the new sets of values, we declare this PC as unsolvable by our algorithm.

This phase is applied to all the PCs that cannot be solved in the Randomization Phase. Note that at the end of the Improvement Phase, each PC may have a different Tabu List. This phase is done by selecting one PC at a time. Therefore, having one Tabu List for each PC is necessary. Randomized values that cannot solve one particular PC may be able to solve another PC in the next iteration of the Improvement Phase.

V. EVALUATION SETTINGS

We evaluate InSPeCT on ten out of the eleven Java programs from the benchmark set that comes with the CarFast tool [32]. CarFast is a technique used to generate high coverage test cases faster by selecting and executing code branches that contain higher numbers of statements earlier.

Table I characterizes the ten Java subjects in terms of their corresponding complexity metrics and their total number of PCs. The first column gives the name of the subject (S) and the second column shows its number of lines of code (LOC). The third column displays the number of classes and methods in each subject (C/M). The fourth (Nested Block Depth/NBD) and fifth (McCabe Cyclomatic Complexity/MCCC [33]) columns show the different complexities of the subjects in the form of average/maximum values. The last column shows the total number of PCs and the number of PCs that are determined by Z3 as unsatisfiable (UNSAT).

Table II shows the characteristics of the PCs from the ten subjects in terms of the minimum, median, and maximum number of branches, variables and various operator.

The experiments are run on a desktop computer running Ubuntu 14 64bit with an Intel Core i7-6700 CPU @ 3.4GHz with 32 GB RAM. InSPeCT is compared with two other state-of-the-art techniques, namely MLB (only one master version) [11] and Z3 (version 4.5.0) [6].We run all models for five consecutive repetitions.

TABLE I
SUBJECTS UNDER TEST WITH THEIR COMPLEXITY LEVELS.

Subject	LOC	**C/M**	**NBD**	**MCCC**	**# of PCs**
tp300	0.3K	4/3	2.5/6	6.3/20	98/97
tp600	600	5/5	2.3/5	10.4/30	189/148
tp1k	1K	18/61	2.2/9	3.8/14	36/34
tp2k	2K	24/49	2.0/5	4.5/13	109/106
tp5k	5K	37/184	2.0/8	5.2/23	106/106
tp7k	7K	38/469	2.2/8	4.3/19	103/101
tp10k	10K	111/765	2.4/8	4.7/23	159/151
tp50k	50K	61/428	4.2/12	22.3/56	191/182
tp80k	80K	96/1.6K	3.4/8	10.7/27	244/184
sp500k	500K	311/2.2K	4/7	34.1/93	146/137

TABLE II
CHARACTERISTICS OF THE PCS

	Min	**Median**	**Max**
# of Branches	3	12	19
# of Variables	1	6	15
# of + operator/PC	0	8	598
# of - operator/PC	0	9	397
# of * operator/PC	0	8	258
# of / operator/PC	0	4	226
# of % operator/PC	0	4	267

Z3 reports that a PC is either solved, UNSAT or UNKNOWN. Z3 may characterise a PC as UNKNOWN if it relies on non-linear arithmetic, as Z3 is highly incomplete in solving non-linear arithmetic equations [24]. We have observed that Z3 may run on some PCs for several hours with no solution. In these cases, Z3 eventually exhausts all the resources. Thus, we decided to timeout the processing of a PC after two minutes. Likewise, InSPeCT times out on a PC after two minutes of processing.

VI. RESULT

Table III shows the total number of solved PCs by InSPeCT. The first column displays the subjects, whereas the second to fourth columns display the statistics about the PCs solved. InSPeCT can solve between 76% and 96% of the solvable PCs out of the ten subjects. It is also fairly consistent in the number of PCs solved with a minimum variance of 0.56 and a maximum variance of 4.56 out of the ten subjects. The fifth and sixth columns of Table III show the mean number of solved PCs in InSPeCT's Randomization Phase (Phase 1) and its variance among the five runs respectively. Phase 1 is able to solve between 32% and 48% of the solvable PCs with a reasonable variance amongst the five runs. On average, InSPeCT is able to solve approximately 44% of the PCs in ten subjects with a reasonable variance.

The seventh through twelfth columns of Table III show the mean number of solved PCs in InSPeCT's Improvement Phase (Phase 2). Columns seven and eight show the mean number of PCs solved in the initial randomized solution generated during Phase 2 and its variance among the five runs respectively. In this stage, few PCs are solved. The main purpose of the solution generated in this stage is to initialize the generation of the solutions for the forthcoming Strategies 1 and 2. Columns nine and ten show the mean number of

TABLE III

TOTAL # OF PCS SOLVED BY INSPECT, AS WELL AS THE # OF PCS SOLVED IN ITS INDIVIDUAL PHASES

Subject	# of PCs Solved in Five Runs			Phase 1		Phase 2 Initial		Phase 2 Strategy 1		Phase 2 Strategy 2	
	Total	Mean	Var	Mean	Var	Mean	Var	Mean	Var	Mean	Var
tp300	90 (93%)	82 (85%)	3.36	39 (40%)	0.4	0.6 (0.6%)	0.2	30 (31%)	15.0	13 (13%)	16.4
tp600	115 (77%)	105 (70%)	3.44	56 (38%)	5.0	1 (0.7%)	1.2	35 (24%)	5.4	12 (8%)	4.8
tp1k	26 (76%)	21 (62%)	2.16	12 (34%)	1.4	0.2 (0.6%)	0.2	7 (21%)	1.6	2 (7%)	1.8
tp2k	102 (96%)	96 (91%)	0.64	46 (43%)	7.8	1.6 (1.5%)	0.6	39 (37%)	9.4	10 (10%)	4.2
tp5k	102 (96%)	99 (94%)	2.24	61 (58%)	1.8	0.8 (0.8%)	0.6	26 (25%)	7.0	11 (10%)	12.4
tp7k	96 (95%)	93 (92%)	0.56	56 (55%)	3.2	0.4 (0.4%)	0.2	26 (26%)	10.8	10 (10%)	6.2
tp10k	132 (87%)	119 (79%)	4.56	79 (53%)	9.0	0.4 (0.3%)	0.6	31 (21%)	11.0	8 (5%)	7.4
tp50k	176 (97%)	170 (94%)	3.44	85 (46%)	1.0	0.8 (0.4%)	0.6	46 (25%)	8.2	39 (22%)	3.4
tp80k	149 (81%)	142 (77%)	0.96	84 (45%)	3.0	0.4 (0.2%)	0.2	38 (20%)	12.2	19 (10%)	2.6
sp500k	116 (85%)	112 (82%)	1.04	43 (32%)	1.0	0.4 (0.3%)	0.6	40 (29%)	8.6	26 (19%)	10.6

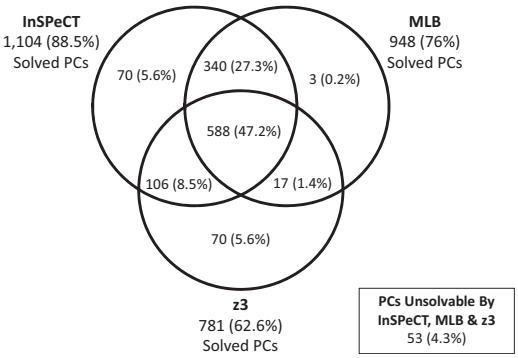

Fig. 2. # of PCs solvable/unsolvable by InSPeCT, MLB & Z3

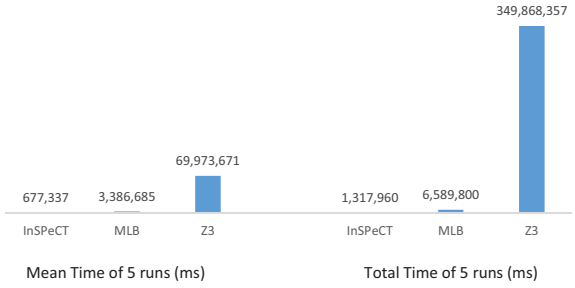

Fig. 3. Time (ms) spent by InSPeCT, MLB & Z3

PCs solved by Strategy 1 of the Improvement Phase and its variance amongst the five runs respectively, whereas columns eleven and twelve show the mean number of PCs solved by Strategy 2 of the Improvement Phase and its variance among the five runs respectively. We observe that Strategy 1 is able to solve on average another 26% of the PCs and Strategy 2 is able to solve on average another 11% of the PCs.

We compare the set of PCs solved by InSPeCT and those solved by the state-of-the-art solvers, MLB and Z3. MLB has been shown [11] to outperform other techniques such as Concolic Walk [25], SPF-CORAL [27], SPF-Mixed [28] and jCute [29]. Z3 is a robust SMT solver used to solve PCs and has been reported to be able to solve more PCs than other SMT solvers such as CVC3 [22] and Yices [23].

Fig. 2 shows the relation between the PCs solved by InSPeCT, MLB and Z3. InSPeCT solves 16% more PCs than MLB and 41% more PCs than Z3. 47.2% of the PCs can be solved by all three approaches. Furthermore, InSPeCT is able to solve 27.3% of the PCs that are solved by MLB but not Z3, and 8.5% of the PCs that are solved by Z3 but not MLB. 5.6% of the PCs can only be solved by InSPeCT and another 5.6% can only be solved by Z3. Only 0.2% of the PCs can be solved by MLB only. When compared with each of the state-of-the-art solvers individually, InSPeCT solves 12.5% more PCs than MLB and 25.9% more PCs than Z3.

Fig. 3 shows the time (mean and total time of five runs) in milliseconds used by InSPeCT, MLB and Z3 to evaluate the PCs. We compare with the mean and total time taken to evaluate the PCs of five runs amongst InSPeCT, MLB and Z3. We observed that InSPeCT is many times faster than both MLB and Z3. The mean time of InSPeCT is approximately 103 $\times$ faster than the mean time of Z3, and the total time of InSPeCT is approximately 265x faster than the total time of Z3. InSPeCT also runs faster than MLB in both the mean and the total time of five runs (approximately 5 $\times$ faster).

We then compare the code coverage amongst InSPeCT, MLB and Z3. Table IV displays the mean (of five runs) line and branch coverage by InSPeCT, MLB and Z3. The first column shows the subject program whereas the second and third columns show the line and branch coverage by InSPeCT. The fourth and fifth columns show the line and branch coverage by MLB whereas the last two columns show the line and branch coverage by Z3. InSPeCT consistently achieves higher line coverage than MLB and Z3 in all the subject programs. Similarly, InSPeCT has higher branch coverage than MLB. Ccompared to Z3, InSPeCT achieves higher branch coverage in all the subjects except for tp1k.

Although InSPeCT solves majority of the PCs, and has the highest PC solving rate (88.5%), a small number of PCs could not be solved by InSPeCT but are solvable by MLB and Z3 (Fig. 2). MLB can solve 3 (0.2%) of the PCs that InSPeCT and Z3 could not solve whereas Z3 can solve 70 (5.6%) of the PCs that InSPeCT and MLB could not solve. Thus, this provides an opportunity to re-evaluate PCs that could not be solved by InSPeCT with MLB and Z3. Since Z3 has a higher percentage (5.6%) of distinct PCs that can only

TABLE IV

MEAN (FIVE RUNS) TEST COVERAGES BY INSPECT, MLB AND Z3.

Subject	InSPeCT		MLB		Z3	
	Line	Branch	Line	Branch	Line	Branch
tp300	80%	85%	76%	80%	78%	81%
tp600	81%	85%	79%	82%	79%	82%
tp1k	66%	63%	64%	60%	66%	64%
tp2k	74%	76%	71%	72%	72%	72%
tp5k	71%	72%	68%	67%	71%	70%
tp7k	70%	75%	67%	70%	69%	72%
tp10k	70%	73%	67%	70%	68%	71%
tp50k	77%	76%	73%	71%	73%	72%
tp80k	70%	67%	68%	65%	69%	66%
sp500k	73%	68%	71%	66%	70%	64%

be solved by Z3 itself, if a PC is unsolvable by InSPeCT, we should pass the PC to Z3 for evaluation before trying MLB (if the PC is unsolvable by Z3). By doing that and if there are new PCs solved by MLB and Z3, but not InSPeCT, it will increase the number of PCs that can only be solved by InSPeCT.

VII. CONCLUSION

We proposed a novel path condition solving technique, InSPeCT, that uses elements of Iterated Local Search and Tabu List. InSPeCT is able to solve a total of 1,104 (88.5%) PCs from ten Java benchmarking subject programs and it is consistent in its solvability performance. When compared to the existing state-of-the-art solvers, MLB and Z3, InSPeCT solves 16% and 41% more PCs than MLB and Z3 respectively. InSPeCT also achieves higher test coverage as compared to MLB and Z3. It is 103 $\times$ faster than Z3 and 5$\times$ faster than MLB, on average.

We plan to investigate building new features to improve InSPeCT's performance, including the use of different data types. The PCs generated by CarFast involve only integer data types. When PCs compose of other data types such as floating numbers, arrays or reference objects, new strategies will need to be applied.

VIII. ACKNOWLEDGEMENT

We thank Julia Lawall in the Whisper group at Inria/LIP6 for the early discussion, writing and proofreading of this work.

REFERENCES

[1] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology, RTI Project*, vol. 7007, no. 011, 2002.

[2] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[3] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 571–572.

[4] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[5] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *FSE*. ACM, 2015, pp. 842–853.

[6] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[7] N. Mansour and M. Salame, "Data generation for path testing," *Software Quality Journal*, vol. 12, pp. 121–136, 2004.

[8] F. Glover and M. Laguna, *Tabu Search*. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[9] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 11, 2012.

[10] M. Harman, "The current state and future of search based software engineering," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 342–357.

[11] X. Li, Y. Liang, H. Qian, Y.-Q. Hu, L. Bu, Y. Yu, X. Chen, and X. Li, "Symbolic execution of complex program driven by machine learning based constraint solving," in *ASE*. New York, NY, USA: ACM, 2016, pp. 554–559.

[12] R. Pargas, M. J. Harrold, and R. Peck, "Test-data generation using genetic algorithms," *Journal of Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 263–282, 1999.

[13] P. J. Van Laarhoven and E. H. Aarts, "Simulated annealing," in *Simulated Annealing: Theory and Applications*. Springer, 1987, pp. 7–15.

[14] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating software engineering as a search problem," *IEE Proceedings - Software*, vol. 150, no. 3, pp. 161–175, 2003.

[15] J. C. Lin and P. L. Yeh, "Automatic test data generation for path testing using GAs," *Information Sciences*, vol. 131, pp. 47–64, 2001.

[16] E. Díaz, J. Tuya, R. Blanco, and J. Javier Dolado, "A tabu search algorithm for structural software testing," *Computers and Operations Research*, vol. 35, no. 10, pp. 3052–3072, Oct. 2008.

[17] O. Sahin and B. Akay, "Comparisons of metaheuristic algorithms and fitness functions on software test data generation," *Applied Soft Computing*, vol. 49, pp. 1202–1214, 2016.

[18] D. Karaboga and B. Basturk, "A powerful and efficient algorithm for numerical function optimization: artificial bee colony (abc) algorithm," *Journal of global optimization*, vol. 39, no. 3, pp. 459–471, 2007.

[19] J. Kennedy, "Particle swarm optimization," in *Encyclopedia of machine learning*. Springer, 2011, pp. 760–766.

[20] R. Storn and K. Price, "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces," *Journal of global optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[21] X.-S. Yang, "Firefly algorithms for multimodal optimization," in *International symposium on stochastic algorithms*. Springer, 2009, pp. 169–178.

[22] C. Barrett and C. Tinelli, "Cvc3," in *Computer Aided Verification*. Springer, 2007, pp. 298–302.

[23] B. Dutertre and L. De Moura, "The yices SMT solver," *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, vol. 2, no. 2, pp. 1–2, 2006.

[24] Z3Prover, "Z3prover github repository issue," 2017, retrieved May 3, 2017 from https://github.com/Z3Prover/z3/issues/887.

[25] P. Dinges and G. Agha, "Solving complex path conditions through heuristic search on induced polytopes," in *FSE*. ACM, November 16-21 2014.

[26] Y. Yu, H. Qian, and Y.-Q. Hu, "Derivative-free optimization via classification," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, pp. 2286–2292.

[27] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu, "Coral: solving complex constraints for symbolic pathfinder," in *NASA Formal Methods Symposium*. Springer, 2011, pp. 359–374.

[28] C. S. Păsăreanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 34–44.

[29] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for C," in *FSE*, vol. 30, no. 5. ACM, 2005, pp. 263–272.

[30] E.-G. Talbi, *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

[31] H. Lourenço, O. Martin, and T. Stützle, "Iterated local search," in *Handbook of metaheuristics*. Springer, 2003, pp. 320–353.

[32] S. Park, B. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie, "Carfast: Achieving higher statement coverage faster," in *FSE*. ACM, 2012, pp. 35:1–35:11.

[33] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.