The Journal of Systems and Software 164 (2020) 110549

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Using likely invariants for test data generation

M. Nosrati, H. Haghighi*, M. Vahidi Asl

Faculty of Computer Science and Engineering, Shahid Beheshti University G.C., Evin, Tehran, Iran

ARTICLE INFO

Article history:
Received 30 July 2019
Revised 16 January 2020
Accepted 11 February 2020
Available online 12 February 2020

Keywords:
Search-based test data generation
Metaheuristic algorithms
Branch likely invariants
Path likely invariants

ABSTRACT

Various approaches have been developed to tackle the problem of automatic test data generation. Among them search-based methods use metaheuristic algorithms to guide search in the input space of the program under test. This paper presents a new approach for improving search-based test data generation methods. This approach is based on learning the relationships between program input values and program parts covered by those values. The learned relationships are used to accelerate achieving test coverage goals. We introduce the concepts of *branch likely invariant* and *path likely invariant* as the basis for the learning method. In addition, we utilize simple predicates (based on some predefined templates) over program input variables to generate better initial candidate solutions, and use the mutation of the mentioned predicates to cover unexplored program parts. The current version of the proposed approach only considers numeric and string input parameters. To evaluate the performance of the proposed approach, a series of experiments have been carried out on a number of different benchmark programs. Through experiments and analysis, we show that the proposed approach enhances the effectiveness of common search-based test data generation methods, in terms of the coverage percentage.

1. Introduction

Test data generation is a challenging task in the software testing process. During this task, for a software under test (SUT), various test data are designed in order to feed into the software to check its conformance to the specified requirements. High quality test data are needed to discover more software failures. For this reason, the problem of generating effective test data, efficiently and quickly, is of great importance. There are several automatic methods for generating test data, each with its own advantages and disadvantages. Among the well-known methods are random testing, symbolic execution, and search-based testing. Random test data generation is easy to implement and fast, but it usually suffers from poor code coverage. In the symbolic execution method, the program is executed with symbolic data, instead of actual data. Then, test data are generated by solving constraints obtained from the symbolic execution of program paths. Some of the main challenges of this method are path explosion, difficulty in handling loops, and solving complex constraints (Anand, 2012).

The focus of this paper is on search-based test data generation methods, which use metaheuristic optimization techniques. The goal of these techniques is to satisfy the specified test goals (such as covering all program branches) by finding appropriate test data. These techniques start from a population of (usually) random initial candidate solutions. The candidate solutions go through several iterations of operations which change the members of population in order to improve the quality of these solutions. Metaheuristic techniques use a fitness function that ranks candidate solutions with respect to the test goals at hand. Search-based methods leverage approaches like evolutionary and swarm intelligence algorithms (e.g., genetic algorithm, ant colony optimization and particle swarm optimization), and simulated annealing. Many of the proposed methods in this area can be reviewed in related papers, such as (Pargas et al., 1999; Tracey et al., 1998; McMinn, 2004; Harman et al., 2012).

The main idea of this research is that, if search-based methods can learn from their past iterations, it may decrease their reliance on sheer chance to generate new candidate solutions, and enable them to generate higher quality solutions at next iterations. This can prevent these methods from performing unnecessary iterations due to low quality candidate solutions which are merely generated based on chance. Consequently, an informed and experience-based search may result in more effective test data while the efficiency of the test data generation method is improved at the same time.

Our solution to implement the mentioned idea is to learn the relationships between covered branches and program input data. These relationships are utilized to generate high fitness input data for the program under test (PUT). The following iterations of the search will use this high fitness data to generate input data that cover uncovered test goals for the PUT.

* Corresponding author.
E-mail addresses: mohammad.nosrati@gmail.com (M. Nosrati), h_haghighi@sbu.ac.ir (H. Haghighi), m.vahidi.asl@gmail.com (M. Vahidi Asl).

https://doi.org/10.1016/j.jss.2020.110549
0164-1212/

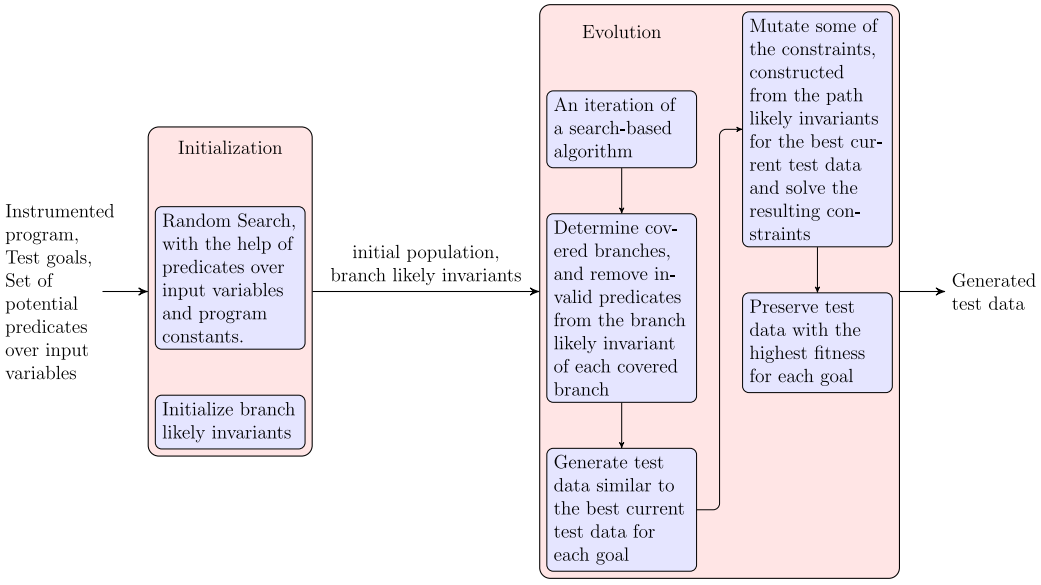

Fig. 1. Summary of the proposed approach.

Fig. 1 shows the summary of the operations in the proposed approach. The proposed approach extends the two main phases of evolutionary search-based algorithms, i.e., the initialization phase, and the evolution phase. At the initialization phase, according to some predefined templates, a set of simple predicates P over program input variables is constructed. These predicates will be used for two different purposes: 1) the generation of the initial population of candidate solutions, and 2) finding *branch likely invariants* for each branch of the PUT. The branch likely invariant of a branch b, denoted by BLI_b, is a set of *likely valid* predicates for b. Likely valid predicates for a branch b are predicates satisfied for all previously generated input data which cover branch b.

The evolution phase of the proposed approach consists of a main loop. This loop involves a normal iteration of a search-based algorithm (such as a genetic algorithm) which evolves the population, followed by a number of additional processing steps. After each iteration of the employed search-based algorithm, for each covered branch b, the branch likely invariant BLI_b is updated. In order to update BLI_b, the predicates associated with branch b are checked against new test data, and those predicates that are not valid anymore, are removed from BLI_b.

At the end of each iteration, the best input data for each test goal are found, and new input data, similar to, but different from these inputs are generated. This is done by constructing constraints from the *BLI* of branches covered by the best found input data, and solving them. These new input data are added to the population of the search algorithm, replacing less fit individuals. Mutation of some of the existing likely invariants, by randomly changing their predicates and solving the constraints, are also performed to cover previously unexplored parts of the program.

To clarify the concept of branch likely invariants, consider program *trivial* in Fig. 2. For this program, which has numeric input variables x and y, at the initialization phase, predicates such as $x = 0, x > y$, and $x = y$ are constructed. Before the evolution phase, we will have $BLI_0 = BLI_1 = \{x = 0, x > y, x = y, x \geq y, x \neq y, \dots\}$. Suppose that at the end of an iteration of the search-based algorithm, the program input $x = 2, y = 1$ has been generated. With this input, branch (0) is covered. Predicates such as $x = y$ and $x = 0$ are invalid according to the generated input and will be removed from BLI_0. At this point, BLI_0 becomes $\{x > y, x \geq y, x \neq y, \dots\}$.

```
def trivial(x, y):
    if (x > y):
        ...                (0)
    elif (x > 0):
        ...                (1)
```

Fig. 2. A trivial program. Branches are shown with (0) and (1).

The approach in this paper can be applied in order to cover a single branch, or a path. A path is a sequence of branches $b_1, b_2, \dots, b_n$, where each pair of adjacent branches in the sequence has a corresponding edge in the control flow graph (CFG) of the program (Offutt and Ammann, 2008). For simplicity, in this paper, we assume that a path is a set of branches (we ignore the order or repetition of branches in paths). Also, we consider a branch as a special path with only one branch. The proposed method is the same for paths with more than one branch, with the difference that the fitness function takes into account all the branches of the path. An input value that covers two branches of a target path has a higher fitness compared to an input value that covers only one branch of that path. For a more rigorous discussion of these notions, refer to Section 4.1.3.

By performing the above-mentioned steps, the proposed approach attempts to improve search-based test data generation methods in a number of ways as follows:

- Initial population improvement: We try to increase the likelihood of finding good candidate solutions in the initial population by using predicates over program variables and source code constants.
- When a suitable candidate solution (i.e., a program input with high fitness) is found, the generation of similar solutions is performed. This is done by constructing and solving path likely invariants.
- More effective coverage of goals by:
  - Generating new input values which are similar to the best available input values in the population, for each goal.
  - Using the mutation of constraints for candidate solutions with the best fitness values.

We experimented the proposed approach on a number of benchmark programs. Our implementation supports numeric and string types. The results show tangible improvements over common search-based methods in terms of the number of iterations and the total coverage of the PUT.

The remainder of this paper is organized as follows. Section 2 starts with an overview of the concepts used in this research. Section 3 presents some of the related works. Section 4 explains the proposed approach. In Section 5 experimental results are given. Section 6 discusses threats to validay. Section 7 includes conclusions and some directions for future work.

2. Background

In this section, we present the required background for this work. In Section 2.1, concepts of test data generation are covered. In Section 2.2, we explain the notion of likely invariants, due to its importance as the basis for the proposed method.

2.1. Test data generation

An important part of software testing is the process of test data generation. For program P which receives n input parameters $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$, for which $x_i \in D_i$, the input domain of the program P is equal to $D = D_1 \times \ldots \times D_n$. The objective of test data generation is to find inputs $\mathbf{x} \in D$, so that by executing P on $\mathbf{x}$, a coverage criterion, such as covering all program branches, is satisfied. Three major techniques exist to generate test data: Random methods, search-based methods and symbolic execution, each of which will be discussed briefly in this section.

In random methods, for a given PUT, random data are taken from a distribution, such as the uniform distribution, over the program's input domain. The advantage of random methods is their ease of implementation and high execution speed. However, their disadvantage is that it is possible that many parts of program code for which the probability of coverage is small are not covered at all, while a considerable number of data are generated to cover the same location (Mariani et al., 2015). There are different approaches for random generation of test data: pure random testing, adaptive random testing and fuzz testing (Mariani et al., 2015).

Another way to generate test data is using symbolic execution (Anand, 2012; King, 1976; Baldoni et al., 2018). In this technique, a program under test is executed with symbolic data (variables) instead of actual data, and a path constraint is generated from every execution path of the program. At any point in the program, the symbolic execution process maintains a mapping σ from program variables to expressions on actual or symbolic values, and a path constraint π which is a formula on symbols to reach to the current program location. To cover an arbitrary branch, the path constraint corresponding to that branch must be given to an appropriate constraint solver.

2.1.1. Search-based methods

Search-based methods use either heuristic or metaheuristic optimization techniques for test data generation (McMinn, 2004; Harman et al., 2012). Some of the algorithms used for test data generation include hill climbing, simulated annealing, genetic algorithms (GA), ant colony optimization (ACO), particle swarm optimization (PSO), and harmony search. In search-based methods, a fitness function is defined based on a coverage criterion, and inputs of the program under test are modified to maximize this function. The fitness function is used to compare the identified solutions with each other.

Hill climbing and simulated annealing are examples of single solution searches, which try to modify and improve a single candidate solution. Hill climbing first randomly selects a solution from the search space and tries to improve this solution. To achieve this goal, the neighborhood of the solution is searched to find a better solution. This process is repeated until no better solution can be found in the neighborhood. The advantage of this method is its simplicity. However, hill climbing may stop at a solution that is only locally optimal (Russell and Norvig, 2016). The identified solution highly depends on the starting point. Harman and McMinn (2007) provide a theoretical and empirical analysis of the application of this method in test data generation. Simulated annealing uses the same procedure, but there is a probability of choosing a solution worse than the previous one, to allow the algorithm to explore more solutions before getting stuck in a local optima (Harman et al., 2012). An example of applying simulated annealing in test data generation is (Tracey et al., 1998).

The particle swarm optimization method optimizes a population of candidate solutions (called particles) by moving them in the search space. Some examples of using PSO in test data generation can be found in Nayak and Mohapatra (2010), Mao et al. (2012a), Lv et al., 2018. Ant colony optimization is another optimization method inspired by pheromone-based communication of real ants which was initially used to find good paths in graphs. However, this algorithm can be used for various optimization problems through either changing the problem into a graph structure or customizing ACO for the problem in hand. Some instances of applying ACO in test data generation are (Li and Lam, 2004; Li et al., 2009; Mao et al., 2012b; Ayari et al., 2007; Sharifipour et al., 2018). Harmony search (Mao, 2014; Sahoo et al., 2016) treats program inputs analogous to musical harmonies and tries to find the best harmony (i.e., program inputs with the best coverage).

An evolutionary method, such as genetic algorithm, is an iterative process, in which a population of solutions, called a generation, is generated at each iteration. Algorithm 1 shows the basic outline of an evolutionary algorithm. At first, a random initial population of candidate solutions is generated. At each generation, the solutions are combined by the crossover operation, and mutated to generate new solutions. The fitness of each of these solutions are evaluated and the most fit individuals are selected to form the next generation. Several examples of the application of these methods in test data generation are (Pargas et al., 1999; Girgis, 2005; Michael et al., 1997; Manaseer et al., 2015; Mishra et al., 2019). The tool EvoSuite uses a genetic algorithm to evolve whole test suites for satisfying the entire coverage criterion (Fraser and Arcuri, 2011; 2014; 2015).

Algorithm 1: Basic outline of an evolutionary algorithm.

```
1 EvolutionaryAlgorithm (P)
  // INITIALIZE POPULATION:
2 population = InitializePopulation()
  // EVOLUTION:
3 while termination_condition not reached do
4 |   Evolve(population)
5 end
```

2.1.1.1. Fitness Function

The fitness function in optimization methods determines how close a solution is to meeting the established goals. For this reason, a better definition of the fitness function has a direct effect on the efficiency and effectiveness of optimization algorithms. For example, in search-based methods to cover a particular branch of the program, the fitness function attempts to minimize two parameters: approach level and branch distance. The approach level evaluates the number of branches to pass through to the destination branch (Harman et al., 2012).

Branch Predicate bp	Branch Distance $BD(bp)$
$boolean$	if $true$ then 0 else K
$a == b$	if $\|a-b\| = 0$ then 0 else $\|a-b\| + K$
$a! = b$	if $\|a-b\| \neq 0$ then 0 else K
$a < b$	if $a - b < 0$ then 0 else $a - b + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $a - b + K$
$A \wedge B$	$BD(A) + BD(B)$
$A \vee B$	$\min(BD(A), BD(B))$
$\neg A$	BD(propagation of negation over A)

Fig. 3. Branch distance for different predicates.

One variable	Two variables	Three variables
$x = a$	$x = y$	$z = ax + by + c$
$x \in \{a, b, c\}$	$x > y, x \neq y$	
$x > a, x <= b$	$y = ax + b$	
$x = 0, x > 0$		
$x = 1, x <= 1$		

Fig. 4. A few possible likely invariants for integer variables *x, y, z*.

Branch distance is defined for branch predicates and indicates how close the inputs are to the execution of a branch (Korel, 1990; Tracey et al., 1998). Fig. 3 shows the branch distance for different types of predicates. The constant *K* is always added if the predicate is not evaluated as *true*. The further away the inputs are from making the predicate *true*, the branch distance's value would be larger. The goal is to minimize this function and get its value closer to zero.

2.2. Likely invariants

Program invariants are predicates that are *true* for every possible inputs of the program. Some of the applications of program invariants are documenting programs, preventing errors, checking hypotheses by using assertions, etc. Predicates that are *true* for all previously observed inputs, but probably not *true* for all possible inputs of a program are called likely invariants. This concept has been introduced in Ernst and Notkin (2000), Ernst et al. (2001). Likely invariants can be extracted by observing the program execution for multiple input values.

There are several predefined templates for checking the existence of a likely invariant. These templates are defined with respect to the number of program parameters and their types. For instance, if *x, y*, and *z* are numeric variables, and *a, b*, and *c* are numeric constants, the invariants shown in Fig. 4 can be considered. These invariants are based on the comparison of variables to each other, linear relationships between variables, comparisons between variables and constants such as 0 or 1, etc).

Dynamic discovery of likely invariants for a given program is done in three phases: 1) Program code is instrumented to trace its dynamic behavior on a given test data. 2) The instrumented program is executed on a test suite to check the validity of possible predefined invariants. 3) At last, those invariants, that are *true* for all the test data in the test suite, are determined as likely invariants.

3. Related works

The existing approaches that combine search-based generation with constraint solvers usually combine random or search-based methods with *symbolic execution*. Some of these works are as follows: Majumdar and Sen (2007) proposed hybrid concolic testing, which interleaves random testing and symbolic execution. Malburg and Fraser (2011) use a genetic algorithm to search for input values of the PUT, and record the path constraint of individuals. They add a special mutation operator for individuals which mutates the path condition of each individual. Sakti et al. (2012) use symbolic execution to replace random generation of data at the initial phase of search based data generation. Baars et al. (2011) enrich the fitness function of search-based data generation methods with statically collected symbolic information. The main difference of these works with our work is that, as though we use constraint solvers, we do not use symbolic execution, or static analysis of the PUT to generate constraints.

Using solutions to constraints constructed from branch likely invariants can be regarded as a method for seeding the initial population. We review some of the seeding strategies in previous works. Fraser and Arcuri (2012) used constants from source code for seeding. Rojas et al. (2016) mention three types of seeding: 1) using constants extracted from source code (numbers and strings), 2) dynamic seeding, or using values observed during test executions, and 3) reusing previously generated test cases. Using likely invariants in our approach as a seeding strategy is different from using dynamic seeding: in dynamic seeding *values* observed during runtime are recorded and used to seed. But in our approach, *relationships* between input parameters are considered (and recorded), and solutions to the constraints constructed from those relationships are used to seed values.

Our proposed method can be used as an extension to many search-based algorithms. Since the proposed method works best with many-objective search algorithms, we review search-based methods which target multiple goals simultaneously. In the Whole Test Suite (Fraser and Arcuri, 2013), a genetic algorithm is used, and an individual of the GA population is a test suite (set of test cases). For fitness function, the sum of all branch distances is used. This work has been extended to archive the best tests seen so far (Rojas et al., 2017). In the Many-Objective Sorting Algorithm (MOSA) (Panichella et al., 2015) each test target (e.g., branch) is an objective. In MOSA, individuals are tests (not test suites). When a new goal is covered, the corresponding test is stored in an archive. DynaMOSA (Panichella et al., 2017) is an improvement over MOSA, where test targets are dynamically selected, guided by the control dependency graph of the program under test. In Many IndependentObjective (MIO) (Arcuri, 2018), a population is maintained for each test target. Once a target is covered, the best test is stored, and the rest are discarded. Our method works by generating similar test inputs to the best found test data, in the hope that these new test data cover uncovered goals. Uncovered goals may be closely related to goals which are already covered. So, generating data similar to, but different from the best data can increase the probability of covering these goals. Therefore, it is most natural to use our proposed method with one of these many-objective search algorithms to consider all goals simultaneously.

4. The proposed method

Search-based test data generation methods have some problems. One of the drawbacks of these methods is that they are *blind* in the sense that they do not have any knowledge of the nature of input values with higher fitness. They attempt to satisfy branch conditions by maximizing the fitness function, and they store the best of the previous data, but they do not have any idea about how the generated input data are correlated with covered goals (except for knowing that these data have higher fitness). Therefore, these methods are incapable of understanding which kind of input values may maximize the fitness function, and thus, they cannot *intentionally* generate input data similar to the most fit ones. Consequently, these methods commonly rely heavily on bare chance. Furthermore, these methods may suffer from the local optima problem (Harman et al., 2012).

Our goal in this paper is to improve search-based test data generation methods and enhance their effectiveness. The first objective of the current research is to devise an approach that enables these methods to learn from past experiences in a more informed and directed manner. By learning the relationships between input variables and coverage goals (expressed in form of predicates over input variables), these methods can generate test data with higher effectiveness. Other objectives of this research include enhancing the initial population of test data, and applying techniques such as the mutation of learned predicates, to mitigate the local optima problem.

In Section 4.1, we present the main algorithm of the proposed method, and discuss its different elements. In Section 4.2, we indicate how we can integrate our method with EvoSuite, to use the power of this framework. The focus of this paper is on numeric and string input parameters. Due to the importance of invariants in the proposed method, in Section 4.3, we present invariants for these types. In Section 4.4, we analyze some aspects of the proposed method, including its performance.

4.1. The proposed test data generation algorithm

Algorithm 2 shows how we can extend a search-based algorithm (Algorithm 1) to use likely invariants. *InitializePopulation* of Algorithm 1 is replaced with *Initialization*, and after *Evolve*, some additional steps are executed. Here, a brief explanation of the different parts of Algorithm 2 is given. In the following subsections, these parts are discussed in more details.

Algorithm 2: The test data generation algorithm using likely invariants.

```
1  TestDataGenerationUsingLI (P, goals, ps)
   Input  : The Instrumented Program P, The Desired Goals
            goals, The Population Size ps
   Output: The set of program inputs covering goals
2  Initialization()
3  while termination_condition not reached do
4  |  Evolve(population)
5  |  UpdateBranchLikelyInvariants()
6  |  foreach best ∈ GetBests(population) do
7  |  |  GenerateInputsSimilarTo(best)
8  |  end
9  |  DiscardSurplusIndividuals(population, ps)
10 end
11 return population
```

The input parameters of the algorithm are the program under test P, the coverage goals *goals* and the population size ps (the number of test data). As mentioned before, the proposed algorithm tries to learn the different predicates which hold for previously generated input values at each branch of the program.

Algorithm 3 shows the Initialization phase of the algorithm. At first, *SimplePredicates* returns the set of simple predicates over program input variables, based on their types, and templates similar to the ones specified in Fig. 4 (some other templates are mentioned in Section 4.3 and 7). These predicates are constructed from program input variables and a set of constants, including common values like 0 or 1, and constant values present in the source code of the program (denoted by PROGRAM_CONSTANTS). For example, for a program with two integer type input variables x and y, and assuming the program source code contains constant value 10, the set of predicates $\{x = 0, x > 0, y = 0, y > 0, x = 1, y = 10, x < 10, x = y, x > y, \ldots.\}$ is returned. These predicates include different combinations of variables x and y, and constant values along with different comparison operators.

Algorithm 3: Initialization phase of the proposed method.

```
1 Initialization (P)
2 preds = SimplePredicates(P, ps)
3 population = InitialPopulation(ps, InputParamTypes(P), preds)
4 foreach Branch br ∈ P do
5 |  BLI_br = preds
6 end
```

It is worth noting that for branches where their coverage is dependent on values of local variables, it is still enough to only consider program input variables, because, ultimately the values of local variables are determined by values of input variables (parameters) of the program. In other words, local variables are indirectly dependent on the values of input variables. For example, in program *triangle_classification* (Fig. 7 in Section 4.1.3), coverage of branches (4) or (6) is dependent on the value of local variable *trian*, which itself is dependent on values of input variables *a, b* and *c*. Therefore, for a PUT with the set of parameters *V, SimplePredicates* constructs predicates with variables from V (and not other variables such as local ones).

In line 3 of Algorithm 3, the initial population of test data is created by calling *InitialPopulation* (see Algorithm 6 in Section 4.1.1), which takes ps, the output of *InputParamTypes*(P) and the set of simple predicates *preds* as arguments. *InputParamTypes*(P) returns a tuple of the types of the program input parameters. The loop of lines 4–6 initializes *BLI* for each branch br with the output of *SimplePredicates*. BLI_{br} is the branch likely invariant for branch br, and is the set of all predicates that are *true* for all generated input values that cover branch br (see Definition 6). For further details on notions such as branch likely invariant and path likely invariant, refer to Section 4.1.4.

The Evolution phase of Algorithm 2 consists of lines 4–10. Each iteration of this loop includes a call to *Evolve*, which is a run of a search-based algorithm, such as GA or PSO (refer to Section 4.1.2). Then, branch likely invariants are updated by calling *UpdateBranchLikelyInvariants* (Algorithm 4) in Line 5. The loop of lines 6–8 generates inputs similar to the best inputs (obtained by calling *GetBests*) and adds these new inputs to the population. In Line 9, the best candidate solutions for each goal are kept and low quality solutions are discarded. At the end, the resulting population is returned as the output of the algorithm.

Algorithm 4: Updating branch likely invariants.

```
1  UpdateBranchLikelyInvariants (population)
2  foreach Individual pi ∈ population do
3  |  foreach br ∈ coverage(pi) do
4  |  |  foreach pred ∈ BLI_br do
5  |  |  |  if pi does not satisfy pred then
6  |  |  |  |  remove pred from BLI_br
7  |  |  |  end
8  |  |  end
9  |  end
10 end
```

The main step for learning branch likely invariants is performed by calling *UpdateLikelyInvariants* in line 5. Algorithm 4 shows the steps of this phase. The loop of lines 2–10 removes those predicates from BLI_{br} which are not *likely valid* anymore. In other words, predicates which are not *true* for program inputs in *population* that result in covering branch br, are removed from BLI_{br}.

Inputs similar to the best inputs are generated by calling *GenerateInputsSimilarTo* (Algorithm 5). In lines 2–3 of this algorithm,

a constraint is constructed from the conjunction of the elements of $PLI_{coverage(input)}$ (i.e., the path likely invariant for *input*, which is defined in Definition 7) and solved using a constraint solver. We consider an input similar to *input*, if it satisfies the predicates of $PLI_{coverage(best)}$. At most F new program inputs are generated for each *input*, where F is a small fraction of the population size. Loop of lines 5–7 mutates the path likely invariant for *best* and solves the resulting constraint, for M times, where M is a small constant.

Algorithm 5: Generating inputs similar to a given input.

```
1 GenerateInputsSimilarTo (input)
2 Construct PLI_coverage(input) // the Path Likely Invariant for
      input
3 similar_to_input =
  GenerateInputsWithConstraint(PLI_coverage(input), F)
4 population = population ∪ {similar_to_input}
5 for M times do
6 |   population =
  |   population ∪ mutate_PLI_and_solve(PLI_coverage(input))
7 end
```

4.1.1. Generating the initial population

The initial population is generated by Algorithm 6. If the program has n inputs, the output of this algorithm is a set of vectors $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$. Instead of just randomly generating program inputs for the initial population, we use two techniques to generate candidate solutions of the initial population. The constant q in line 5 of Algorithm 6 specifies the probability of choosing either techniques.

Algorithm 6: Populating the initial generation.

```
1  InitialPopulation(ps, param_types, preds);
2  init_population = ∅
3  for i ← 1, ps do
4  |   input = New tuple with size param_types.size
5  |   if random() < q then
6  |   |   for j ← 1, param_types.size do
7  |   |   |   constants_from_program =
   |   |   |   {val|val ∈ PROGRAM_CONSTANTS ∧
8  |   |   |   |   type(val) = param_types[j]}
9  |   |   |   if constants_from_program ≠ ∅ then
10 |   |   |   |   input[j] =
   |   |   |   |   random_choice(constants_from_program) with a
   |   |   |   |   small probability p or a random value with
   |   |   |   |   probability 1 − p
11 |   |   |   end
12 |   |   |   else
13 |   |   |   |   input[j] = A random value
14 |   |   |   end
15 |   |   end
16 |   end
17 |   else
18 |   |   constraint = random_choice(preds)
19 |   |   input = solve(constraint, param_types)
20 |   end
21 |   init_population.add(input)
22 end
23 return init_population
```

The first technique is to generate program inputs randomly, or by utilizing the constant values in the program text. Fraser et al.

```
def gcd(a, b):
    if a < 0:
        a = −a                (0)
    if b < 0:
        b = −b                (1)
    if a == 0 or b == 0:
        return 0              (2)
    while b > 0:
        tmp = a % b           (3)
        a = b
        b = tmp
    return a
```

Fig. 5. The gcd program.

mentioned the use of constant values for the generation of initial population from the program source code in Fraser and Arcuri (2012). The use of constant values may highly increase the probability of covering some branches. Because there is a high chance that these constants (or a variation of them) appear in some conditions, therefore, the chance of satisfying the conditions and covering the relevant branches is increased. In Algorithm 6, the loop of lines 6–15 probabilistically selects values for program input variables, from the set of program constants or completely random values.

Another technique that we apply to increase the quality of the initial population, is to leverage the power of unknown existing relationships between program input variables, as well as relationships between program input variables and constants. Before checking the actual program behavior, we can try to randomly sample from possible predicates among program arguments, and solve them. This technique can be considered as a generalization of the first technique, too. Instead of just inserting the constant values in the initial population, we construct different predicates over program variables and constants, solve them and add the solutions to the initial population. This is done in lines 18–19 of Algorithm 6. As an example, if the program has two numeric arguments x and y, we can sample from possible predicates over two integers (such as $x = y$, or $x < y$, etc.) and solve them. For instance, in the *gcd* program shown in Fig. 5, the probability of randomly generating an input $\langle a, b \rangle$ such that branch (2) is covered (i.e., generating $\langle a, b \rangle$, where $a = 0 \vee b = 0$), is $\frac{1}{2^{31}}$ (where a and b are 32 bit numbers). But if Algorithm 6 is used, since the constant 0 exists in the program text, by using predicates such as $a = 0$, the probability of covering the branch (2) will become extremely high.

4.1.2. Search-based algorithm iteration

The method *Evolve* (called in line 4 of Algorithm 2) performs one iteration of a search-based algorithm. Any metaheuristic algorithm, including GA or PSO, can be used. An iteration of these algorithms is executed at each iteration of the main loop of Algorithm 2. For example, Fig. 6 shows the case where GA is used.

The input of the algorithm in Fig. 6 is the current population. In line 4, the crossover operation is performed. For a binary-coded test data vector, the crossover operation can be defined as follows: one slice from the beginning of the bit stream representation of the first vector (as the first parent) and one slice from the end of the bit stream representation of the second vector, as the second parent, (from the same location selected for the end of the first vector) are selected, and the child vector is generated from the combination of these two bit streams. In line 5, the mutation operation is performed. The mutation operation can be defined as reversing some bits with a small probability. For the Python implementation of the proposed method, we use binary-coded GA for chromosomes. In the examples of this work, for clarity, we use actual values of numbers, instead of their bit-stream representations.

```
1 Evolve(population);
2 new_population = ∅
3 Select a predefined number of pairs from population
4 Apply the crossover operator on all pairs and add the offsprings to new_population
5 Apply the mutation operator with a small probability on the members of
    new_population
6 population = population ∪ new_population
```

Fig. 6. An iteration of GA.

```
def classify_triangle(a, b, c):
    if a <= 0 or b <= 0 or c <= 0:
        return 'INVALID'                                    (0)
    trian = 0
    if a == b:
        trian = trian + 1                                   (1)
    if a == c:
        trian = trian + 2                                   (2)
    if b == c:
        trian = trian + 3                                   (3)
    if trian == 0:
                                                            (4)
        if a + b <= c or a + c <= b or b + c <= a:
            return 'INVALID'                                (5)
        else:
            return 'SCALENE'                                (6)
    if trian > 3:
        return 'EQUILATERAL'                                (7)
    if trian == 1 and a + b > c:
        return 'ISOSCELES'                                  (8)
    elif trian == 2 and a + c > b:
        return 'ISOSCELES'                                  (9)
    elif trian == 3 and b + c > a:
        return 'ISOSCELES'                                  (10)
```

Fig. 7. The triangle classification program.

In case of PSO, at each iteration, the population of candidate solutions, here called particles, is moved in the search space according to a formula over particles' velocity and position. In other words, instead of mutation and crossover operations, solutions are generated by changing particles' positions. Each particle's movement is guided by its best known position, in addition to the global best known position. Since the proposed method is independent of the chosen search-based algorithm, other algorithms can be used as well.

4.1.3. Fitness function

As stated before, in order to have a high quality search-based algorithm, having a proper fitness function is of great importance. The fitness function must be such that it has high values for candidate solutions close to the target goal, and the farther a solution gets from the target goal, its value must be smaller.

Here, we define the fitness functions for branch and path coverage goals. In order to be more precise, we need some definitions beforehand. We represent vectors such as program inputs, target paths, and coverage vectors using the angle bracket notation. The ith element of a vector $\mathbf{v}$ is represented by $\mathbf{v}_i$. In this subsection, we use n as the number of branches of the program under test P, and D_P is its input domain. P's branches are indicated by numbers 0 to $n-1$.

Definition 1. For a program P with n branches indexed from 0 to $n-1$, with input domain D_P, and program input vector $\mathbf{x}$, we define the coverage vector $coverage(\mathbf{x})$ as follows:

$$\begin{aligned}&\forall i \in 0\,..\,n-1, \mathbf{x} \in D_P.\\ &(\text{branch } i \text{ is covered when running } P \text{ on}\\ &\mathbf{x} \Rightarrow coverage_i(\mathbf{x}) = 1) \ \wedge\\ &(\text{branch } i \text{ is not covered when running } P\\ &\text{on } \mathbf{x} \Rightarrow coverage_i(\mathbf{x}) = 0)\end{aligned} \tag{1}$$

If running P on $\mathbf{x}$ results in the coverage vector $coverage(\mathbf{x})$, we say the covered path is $coverage(\mathbf{x})$. In this paper, we only consider the branches covered in paths, and the order and repetition of branches are ignored.

Definition 2. The target coverage vector $\mathbf{T}_{1\times n}$ shows the desired branches to cover. The range of this vector is $\{0, 1\}$ and $G = \{i | \mathbf{T}_i = 1\}$ indicates that the goal is to cover the branches with indices $i \in G$. For a branch coverage goal, only one of the entries of this vector that corresponds to the target branch is set to 1.

Definition 3. Let $\mathbf{T}_{1\times n}$ be the target coverage vector. We define the branch coverage similarity for branch i with branch predicate bp, for input $\mathbf{x}$ as:

$$s_i(\mathbf{x}) = \begin{cases} 0 & \mathbf{T}_i = 0 \\ 1 - N(BD_{\mathbf{x}}(bp) + AL_{\mathbf{x}}(i)) & \mathbf{T}_i = 1 \end{cases} \tag{2}$$

The branch distance $BD_{\mathbf{x}}(bp)$ is the same as shown in Fig. 3. $AL_{\mathbf{x}}(i)$ is the approach level of input $\mathbf{x}$, with respect to branch i. The function N normalizes its parameter to a scale of 0 to 1. For N, we can use EvoSuite's normalization function: $N(dist) = \frac{dist}{dist+1}$ (Fraser and Arcuri, 2013).

Definition 4. For target vector $\mathbf{T}_{1\times n}$, we define the path coverage similarity vector $\mathbf{S}(\mathbf{x})$ for input $\mathbf{x}$ as:

$$\mathbf{S}(\mathbf{x}) = [s_i(\mathbf{x})]_{1\times n}, \text{for } i \in 0\,..\,n-1 \tag{3}$$

Finally, by measuring the cosine similarity between $\mathbf{S}$ and $\mathbf{T}$, we obtain the closeness of an input to cover the desired path:

Definition 5. The fitness function for input $\mathbf{x}$ is defined as:

$$f = \frac{\mathbf{S}(\mathbf{x}).\mathbf{T}}{|\mathbf{S}(\mathbf{x})||\mathbf{T}|} \tag{4}$$

Example 1. Fig. 7 shows the triangle classification program in Python. This program receives three integers *a, b*, and *c*, and returns the triangle type that these three numbers make. Possible outputs are INVALID, SCALENE, ISOSCELES, and EQUILATERAL.

In this example, for *BD* formulas in Fig. 3, we choose $K = 1$. For the input $\langle 1, 2, 3\rangle$, according to Definition 4, the path coverage similarity vector is obtained as follow (due to space limits, we omit the computations):

$$\langle 0.998, 0.998, 0.997, 0.998, 1, 1, 0, 0, 0, 0, 0, 0\rangle,$$

where elements which are equal to 1 show branches that are covered by this input and other non-zero elements show how close this input is to covering the corresponding branch. Zero elements show that the corresponding branches are not covered, and no branch distance information is available for them, because their corresponding conditions were never evaluated during the execution of the program on this input. So, if the target coverage vector is $\langle 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0\rangle$, the fitness will be equal to 0.866.

4.1.4. Learning the relationships between inputs and covered branches

One of the main aims of this research is devising a method to learn the relationships between inputs and covered branches.

```
def example(x, y):
    if x>= 0:
        ...                (0)
        if y >= 10:
            ...            (1)
    ...
```

Fig. 8. An example program to show how *BLI* and *PLI* are constructed.

These learned relationships are used to generate new data for previously covered branches, as well as to discover input data for new branches or paths. To achieve this goal, we define the notion of branch likely invariants and path likely invariants.

Definition 6. For a given program with n branches indexed from 0 to $n-1$, and a set of generated test data *TS*, the branch likely invariant BLI_i is a set of predicates that are valid for all test data in *TS* covering branch i.

Definition 7. For a given program with n branches indexed from 0 to $n-1$, and a set of generated test data *TS*, the path likely invariant PLI_p is the union of predicates of branch likely invariants of all branches of path p.

$$PLI_p = \bigcup_{i \in 0..n-1 \text{ where } p_i=1} BLI_i \tag{5}$$

We also define the path likely invariant for program input $\mathbf{x}$ as $PLI_{coverage(\mathbf{x})}$.

In order to use *BLI* and *PLI*, these sets are converted to a constraint and are fed to a constraint solver.

Definition 8. For branch b with branch likely invariant BLI_b, $Constraint(BLI_b)$ is defined as $\bigwedge_{pred \in BLI_b} pred$. For path p with path likely invariant PLI_p, $Constraint(PLI_b)$ is defined as $\bigwedge_{pred \in PLI_b} pred$.

Example 2. Consider the program in Fig. 8 and the following set of test data: $\{\langle 10, 0\rangle, \langle 6, -7\rangle, \langle 7, -5\rangle, \langle 0, 10\rangle, \langle 0, 0\rangle, \langle -5, 6\rangle, \langle 12, 12\rangle, \langle 7, 6\rangle\}$. A possible branch likely invariant for the branch (0) can be the set $\{x \geq 0, \ldots\}$. The path likely invariant for the path $\langle 1, 1, \ldots\rangle$ (i.e., the path consisting of branches 0 and 1, according to Definition 2) is $PLI = \{x \geq 0, y > 0, y \geq 0, y \geq 10, \ldots\}$ (according to Definition 7). In addition, the corresponding constraint for the path likely invariant of this path is $Constraint(PLI) = x \geq 0 \wedge y > 0 \wedge y \geq 0 \wedge y \geq 10 \wedge \ldots$.

We obtain the relationships between inputs and covered branches by running the program and evaluating the inputs in each covered branch. This is performed in line 5 of Algorithm 2. The method that we use to learn these relationships is as follows: we consider a collection of initial predicates for input variables based on their types and templates similar to the ones specified in Fig. 4, and initialize *BLI* for all branches with this set. For each input that covers branch i, we evaluate all the remaining predicates in BLI_i. If a predicate does not hold for an input, it will be removed from the collection (line 6 of Algorithm 4). What remains at the end is the learned relationships, i.e., the branch likely invariant for branch i.

4.1.5. Generating inputs similar to the best input

At this stage (i.e., lines 6–8 of Algorithm 2), for each given goal (e.g., each branch), the best input found so far (based on the fitness function) is considered. Then, using the path likely invariant of this input, several similar inputs are generated. In this way, we will increase the quality of the whole population at a much higher rate and we will converge to the final solutions more quickly.

For generating similar inputs, we use a constraint solver, such as Microsoft Z3 (De Moura and Bjørner, 2008). In Algorithm 2, *SimplePredicates* determines the templates for atomic predicates. These templates must be simple enough for the used constraint solver. For example, they can be linear predicates.

The default configuration in Z3 results in the generation of the same values for one constraint. In order to diversify the solutions, some changes should be applied to the constraint, to prevent the constraint solver from generating the same outputs. One way is to explicitly exclude previously found solutions for a specified constraint. Let S be the set of already found solutions for a constraint C, and $\mathbf{v} = \langle v_1, \ldots, v_n\rangle$ be the vector of program input variables. For each solution $\mathbf{x} = \langle x_1, \ldots, x_n\rangle \in S$, we define the predicate $\eta(\mathbf{x})$ as $\bigwedge_{i=1}^{n} v_i = x_i$. Now, the constraint that is given to the constraint solver is:

$$C \wedge \neg(\bigvee_{\mathbf{x} \in S} \eta(\mathbf{x})) \tag{6}$$

When the number of already found solutions increases, the cost of solving this constraint goes up. Another technique for diversifying solutions is to partition the domain of one or more input variables. For example, for the input variable x_i, if it is of the numeric type, we can add conditions such as $x_i \in [a_j, b_j]$, $j \geq 1$, for some numbers a_j and b_j ($a_j < b_j$), where $\bigcup_j [a_j, b_j]$ is the entire set of values x_i can take. These conditions partition the domain of x_i into a set of intervals. This technique performs much better compared to the first one, but the solutions lose the contiguity that the first technique provides (while this contiguity is desired for covering parts of some programs). For the best effect, we should use a combination of both techniques. In other words, we exclude a sample of already found solutions, as well as partitioning some of input variables. Algorithm 7 shows the input generation function through constraint solving. The input parameter *constraints* is a set of predicates, and n is the number of desired solutions. We show all the additional constraints discussed here, to avoid duplicate solutions (as *additional_constraints*).

Algorithm 7: Generating inputs using constraint solving.

```
1 GenerateInputsWithConstraint(constraints, n);
2 new_inputs = ∅
3 for n times do
4     new_inputs = new_inputs ∪ solve_constraint(⋀_{c∈constraints} c ∧
      additional_constraints)
5 end
6 return new_inputs
```

Example 3. For the triangle program (Fig. 7), suppose we want to cover the "EQUILATERAL" branch (i.e., branch (7)), and the best input found so far is $\langle 19, 11, 18\rangle$, which can have the following path likely invariant:

$$\{b > 0, a \geq 0, c \geq 0, c > 0, b \geq 0, a > 0\}$$

By solving the constraints obtained from this path likely invariant (i.e., $b > 0 \wedge a \geq 0 \wedge c \geq 0 \wedge c > 0 \wedge b \geq 0 \wedge a > 0$), inputs such as $\langle 17, 15, 15\rangle$ and $\langle 19, 4, 1\rangle$ are obtained and added to the population. In the next iteration, the search algorithm finds the input $\langle 17, 15, 15\rangle$ as the best input. Its path likely invariant can be equal to:

$$\{b > 0, a \geq 0, c \geq 0, c > 0, b \geq 0, a > 0, b = c\}$$

By solving the corresponding constraints, inputs such as $\langle 18, 20, 20\rangle$ and $\langle 17, 18, 18\rangle$ are obtained and added to the population. At the next step of the search algorithm, by applying crossover and mutation operations, input $\langle 18, 18, 18\rangle$, which has fitness 1 is found. The path likely invariant for this input is:

$$\{b > 0, a \geq 0, c \geq 0, c > 0, a = c, b \geq 0, a = b, a > 0\}$$

From this point on, it is extremely easy to generate many inputs covering the "EQUILATERAL" branch. In fact, it is possible to generate *all* such inputs by simply solving the learned constraint, i.e., $b > 0 \wedge a \geq 0 \wedge c \geq 0 \wedge c > 0 \wedge a = c \wedge b \geq 0 \wedge a = b \wedge a > 0$.

4.1.6. Mutating path likely invariants

As a simple example, suppose there exists an $if - then - else$ statement, with the branch condition $x > 0$ that is already covered. Also, suppose the input $x = 1$ is the best currently found solution for the *then* branch (denoted by *best*), and its *PLI* is $\{x > 0\}$. If we try to mutate this path likely invariant, we would get constraints such as $x \leq 0$ which is exactly the condition needed to cover the *else* branch. That is what *mutate_PLI_and_solve*, in line 6 of Algorithm 5, does: mutating obtained path likely invariants for *best*, by changing the comparison operators, operands or constants of one or more atomic predicates of the constraint, and solving the resulting constraints to achieve more diversity in candidate solutions, and increasing the probability of covering new unexplored locations.

4.1.7. Multi-goal search

As shown in works such as (Lakhotia et al., 2007; Ferrer et al., 2012), it is more efficient and effective to search for covering multiple (perhaps conflicting) objectives at the same time. For example, the test data generation tool EvoSuite uses an evolutionary search approach which evolves the whole test suite to satisfy the entire coverage criterion (Fraser and Arcuri, 2013).

Though the proposed method can be used with any search-based algorithm (with the basic outline of Algorithm 1), it is best to use it with multi-objective search methods such as Whole Suite, MOSA, or MIO.

For a multi-objective algorithm such as MIO, the population is partitioned into multiple parts, each for a particular goal. Regarding all-branches coverage, for a program with n branches, the set *goals* will be defined as a set of $goal_j = [b_i]_{1\times n}$ (for $i, j \in 0.n-1$) vectors, where $(i = j \Rightarrow b_i = 1) \wedge (i \neq j \Rightarrow b_i = 0)$. We can partition the population (denoted by *population*), into n parts, with size $\frac{|population|}{n}$, each subpopulation denoted by $population_j$ and containing the best found candidate solutions for one branch. At the end of each iteration of Algorithm 2 (lines 6–8), we generate inputs similar to the input with the highest fitness in each partition (obtained by *GetBests*), and keep the most fit $\frac{|population|}{n}$ solutions for each goal (line 9). In case of PSO and similar algorithms, before starting the first iteration, each particle is assigned to a goal. Each partition has its own global and best position, and particles are divided into clusters, in which they move around their respective best positions.

4.2. Integration of the proposed method with EvoSuite

EvoSuite (Fraser and Arcuri, 2011; 2014; 2015) is an excellent test data generation tool that uses evolutionary methods to generate tests for Java programs. In this section, we explain how it is possible to integrate the proposed method with EvoSuite in order to use the power of this open-source framework. We only changed the part of EvoSuite which generates values for parameters with supported types (numeric, boolean, and string types, in our case). The goal selection and fitness calculation mechanisms, and the overall algorithm, including the method of evolution or the underlying multi-objective method used in the algorithms employed by EvoSuite remain unchanged.

EvoSuite uses a sequence of Java statements as its chromosomes. For example, for a class *C*, which has a method `void m(int x, int y) { ... }`, a simple EvoSuite chromosome can be like this:

```
C c0 = new C();
int int0 = 0;
int int1 = 1;
c0.m(int0, int1);
int int3 = 10;
c0.m(int1, int3);
```

During the initial population generation and the evolution phases, the GA algorithm uses a class named `TestFactory` to generate chromosomes. This class adds random object creation and method calls for the class under test. When a method call is added, `TestFactory`'s `satisfyParameters` is called for that method call, and the required parameters are generated (or maybe reused, if any exist), and statements for defining those parameters are inserted before the method call.

In order to integrate our method with EvoSuite, we changed the `satisfyParameters` function to probabilistically satisfy parameters with supported types. The pseudo-code in Fig. 9 shows how this can be done.

`generateValuesUsingLIsFor` generates values similar to an input $\mathbf{x}$ which is chosen in one of these two cases: either one of the best currently found inputs (in EvoSuite, chromosomes with fitness value of 0 are the best), or an input which has a non-zero distance to cover one of the uncovered goals (in our cases, branches). It then generates a similar input to the chosen input $\mathbf{x}$ by one of these methods:

- Finding the *PLI* of $\mathbf{x}$, and creating a different value $\mathbf{x}'$ with the same *PLI*.
- Mutating the *PLI* of $\mathbf{x}$, and solving the mutated constraint.
- Mutating the values of $\mathbf{x}$, directly.
- Using previously found parameter values.

With this code, we can use our algorithm for programs with all kinds of input parameters, but only generate values for supported types (numeric and string types in our case). EvoSuite will generate values for unsupported types.

4.3. Invariants for numeric and string data types

The proposed method does not put any restraint on the invariants used, except that constraints (constructed from them) be solvable by a solver. However, using too many invariants may be actually detrimental, because some of them can be always true, but useless. For example, for integer parameter x, we can use invariants $x \neq a$, for any integer number a, and these invariants will be *true* for most branches; but they are hardly useful, and just degrade the performance by wasting time for validating these invariants and solving useless constraints. For numeric parameters, invariants as shown in Fig. 4 may be sufficient in many cases. But, an implementer of the proposed method may choose to add more invariants, as long as they remain solvable. For example, for numeric parameters x, y, and z, and constant a, these invariants can be added (Some of the following invariants were introduced in Ernst and Notkin (2000)):

- Remainder is zero (or is not): $(x \mod a) = 0$ or $(x \mod a) \neq 0$
- $x = |y|$
- $x = -y$, $x = 2y$, $x = \frac{1}{y}$, $x = y^2$, $x = \sqrt{y}$
- $z = \min(x, y)$, $z = \max(x, y)$
- $z = x \times y$, $z = \frac{x}{y}$, $z = \frac{y}{x}$

Some useful invariants can be obtained by using the pools of constant or dynamic seeding (discussed in Section 3). For example, for a constant d from the pools, we can have constraints such as $x = d$, $x > d$, $x + y = d$, etc.

```
satisfyParameters(method, parameters, parameterTypes) {
    if (random() < LI_PROBABILITY) {
        if parameterTypes contain SUPPORTED_TYPES {
            generated_values = generateValuesUsingLIsFor(method)
        }
    }

    for (p ∈ parameters) {
        if value is generated for p in generated_values {
           p.value = value
        }
        else
            the normal course of satisfyParameters
    }
}
```

Fig. 9. Customized `satisfyParameters` of EvoSuite for the proposed method.

For strings S and S', we consider these invariants ($|S|$ denotes the length of S):

- S is null, or S is empty ($|S| = 0$).
- $|S| = |S'|$
- S and S' are equal: $|S| = |S'| \wedge \forall i \in 0..|S|.S[i] = S'[i]$. Also, S and S' are equal with cases ignored.
- S is a substring of S'.
- S contains S', S starts with S', or S ends with S'.
- S is the reverse of S'.
- S has only ASCII, or non-ASCII characters, or both.
- S contains character c.

For strings, string solvers such as Z3-str (Zheng et al., 2013) can be used. In Section 7, we mention some invariants for arrays and general objects.

4.4. Analysis of the method

In this section we discuss the ways our proposed method can improve the search-based (especially metaheuristic) methods, and how it tries to avoid the problems of current test data generation methods as much as possible.

4.4.1. Improvements on search-based methods

The proposed method attempts to improve search-based test data generation by increasing the effectiveness of each iteration of search by a variety of techniques:

- **Starting with an initial population which takes into account different relationships of parameters**. By sampling the possible relationships among program input variables, and generating data that satisfies these relationships, we try to increase the probability of starting with an initial population that takes into account the different relationships that parameters can have with respect to each other. Table 9 shows an improvement on the overall fitness of the initial population when using likely invariants.
- **Generating data similar to the most fit data**. The main objective of the proposed method is to improve the effectiveness of the iteration by using past experiences. If an individual (solution) with a high fitness has been found, we would want to generate similar solutions to increase the probability of finding individuals that cover the desired goals.
  Apart from specific operators of search-based algorithms (such as crossover and mutation in GA), random sampling of data plays an important role in search-based algorithms. Therefore, improving the random sampling part of search-based algorithms, has a direct impact on the performance of these algorithms. In Proposition 1, we show that the proposed algorithm improves the performance of random data generation for dependent branches. Before presenting Proposition 1, we define the notion of branch-dependency.

  Definition 9. If the execution of branch b_2 depends on the execution of branch b_1, we say b_2 is branch-dependent on b_1 (denoted by $b_1 \gg b_2$).

  For example, in Fig. 8, (0) $\gg$ (1).

  Proposition 1. *The proposed approach improves the performance of branch coverage, for many-objective random data generation algorithms.*

  Proof. In a chain of branch-dependency, for example $b_0 \gg b_1 \gg \cdots \gg b_n$, the domain of program inputs covering a branch gets smaller and smaller as we go down the chain. For a program with input domain D, if we denote the subdomain corresponding to branch b_i by D_i, we will have $D_0 \supseteq \cdots \supseteq D_n$.
  We denote the parts of the proposed algorithm, relevant to the generation of new solutions using the likely invariants by the operator λ. Suppose that λ is able to produce different samples from a particular D_i. Regarding branch b_{i+1}, since D_{i+1} is a subset of D_i, the probability of sampling from this domain by λ is equal to $\frac{|D_{i+1}|}{|D_i|}$.
  If I is the number of samplings until b_{i+1} is covered, I follows a geometric distribution with parameter $\frac{|D_{i+1}|}{|D_i|}$. Therefore, the average number of samplings to cover this branch will be

  $$\frac{|D_i|}{|D_{i+1}|} \tag{7}$$

  The probability of randomly sampling from D_{i+1} is $\frac{|D_{i+1}|}{|D|}$, and therefore, the average number of samplings will be $\frac{|D|}{|D_{i+1}|}$. If $|D| = \alpha|D_i|$, the number of needed samplings to cover b_{i+1} will be reduced by a factor of α (The larger α is, this reduction becomes more tangible). Thus, the proposed algorithm leads to faster coverage of dependent branch b_{i+1} compared to random data generation algorithms. □
- **Mutating the obtained path likely invariants**. This is helpful in cases where some changes in obtained constraints for cover-

```
def ex_1(x, y):
    if 2 ** x > 100:
        ...                          (0)
    if x ** 2 + y ** 2 > 25:
        ...                          (1)

def ex_2(x, y, z):
    if 2 ** x > 2 ** y and log(y) == log(z):
        ...                          (0)

def ex_3(x):
    w = 0
    for i in range(1, x+1):
        w += 1
    if w > 10:
        if (...):                    (0)
            ...
```

Fig. 10. Showing situations where constraints constructed by the dynamic analysis might be easier to solve.

ing a branch leads to achieving constraints that their solutions satisfy other branch predicates.

4.4.2. The issue of constraint solving

Since some aspects of our method depend on constraint solving, it may be concerning that our method has the problems of symbolic execution methods related to the capabilities of constraint solvers. While the capabilities and performance of constraint solvers make a real issue for our method too, there is a significant difference between symbolic execution methods and ours. The main difference in contrast to symbolic execution and constraint based test generation techniques (such as DeMilli and Offutt (1991)), is that in the case of our proposed method, constraints are all dynamically constructed from observing values of actual inputs, but in the symbolic execution methods, they are obtained by statically analyzing the source code. In case of dynamic symbolic execution, although the program is run with actual data, it still constructs constraints by statically analyzing the code, but replacing some program variables with known values.

Some benefits can be achieved from dynamic construction of constraints instead of static construction:

- **The ability to obtain simpler constraints** - Observing data and deriving constraints from them may lead to simplified versions of constraints that could be constructed from statically analyzing the source code. Sometimes, it is sufficient to get an approximate constraint (from dynamic analysis) to generate test data.
  The dynamic constraints are constructed from a predefined set of predicate templates, which all of them are ensured to be of the types that are easily solvable by the selected constraint solver. For instance, if a constraint solver is able to solve linear constraints, only predicates which are linear with respect to all variables are considered. Therefore, the dynamically constructed constraints will be a combination of linear predicates. This approximation makes the constraints easier to solve, compared to the ones which would be constructed by the static analysis.
  Fig. 10 shows some examples, which while being somewhat contrived, illustrate the idea. For branch (0) in method *ex*_1, instead of solving the complex constraint $2^x > 100$, imprecise constraints such as $x > 7$ can be dynamically obtained and solved. For branch (1), the dynamic constraint may be $x > 4 \wedge y > 4$, which is a boolean combination of atomic linear predicates. In rarer cases, it may be possible to obtain simpler, but equivalent constraints. For example, in Fig. 10, the symbolic engine is not able to solve the constraint for branch (0) in method *ex*_2. In the proposed method, since only data values are considered in the branches, the equivalent constraints, i.e., $x > y \wedge y = z$, is easily achieved and solved.
- **Independence from execution path** - Statically analyzing the source code to obtain constraints can lead to very long and complex constraints, because it has to consider every change in the value of a variable in the course of a program path to a particular program location. The existence of loops exponentially worsens this situation. This point ultimately comes down to the point mentioned above too (i.e., getting simpler constraints). But since the source of difficulty in these constraints is different (the constraints are dependent on program paths and loops), it is worth being discussed separately.
  To show the idea, consider the method *ex*_3 in Fig. 10. The loop in this example just calculates $w = \sum_{i=1}^{x} 1 = x$, and the *if* branch is equivalent to $x > 10$. But it is obvious that the static analysis of this program leads to obtaining a far more complex constraint, opposed to just observing the values of x in branch (0).

The simplifications obtained by the dynamic analysis are also useful for sub-programs. Programs may, in their conditions, require the execution of a function or subprogram, which cannot be directly analyzed and solved by the symbolic methods. In the proposed method, it is not necessary to solve these constraints, and only the relationships that are required between the input variables to reach a branch are considered. In addition, if local pointers are used, there will be a problem of pointer analysis in the symbolic methods that is not a concern in the proposed method.

4.4.3. Performance and scalability

The initialization phase of Algorithm 2, as indicated in Algorithm 3, performs three operations: generating simple predicates, generating the initial population, and assigning these simple predicates to each branch. The first and last operations are not an issue for performance. The first operation generates different *k*-combinations (for small *k*s such as 1, 2 or 3) of program parameters, and combines these combinations with suitable operators (by type). The last operation is just some assignments. The performance of the second operation, i.e., the generation of initial population, as indicated in Algorithm 6, is not very different from typical search-based algorithms. At this stage, the constraints are simple enough to not require a real constraint solver to be solved. For example, $x = y$ is solved by randomly choosing a random value and assigning that value to both x and y; $x < y$ is solved by randomly choosing two different values and assigning the smaller one to x.

Algorithm 6 generates the initial population without updating the branch likely invariants, which needs running the PUT and getting feedback from the program under test. However, it is possible that we perform these operations in the initialization phase too. In this case, the overhead will be similar to the overhead of an iteration of the proposed method in the evolution phase.

The evolution phase of Algorithm 2 adds two major operations to the iteration of a typical search-based algorithm: *UpdateBranchLikelyInvariants* and *GenerateInputsSimilarTo*. *UpdateBranchLikelyInvariants* checks if the remaining predicates in branch likely invariants are still valid for new inputs. Checking predicates in branch likely invariants (such as $x < y$ with a new $\langle x, y \rangle$) is easy, and the costs are negligible. In addition, most of these predicates are contradictory with respect to each other (such as $x > y$, $x = y$, and $x < y$), and thus, most of them are falsified and removed as soon as a branch is covered.

The real overhead is caused by *GenerateInputsSimilarTo* which performs constraint solving. Here, we estimate the overhead of *GenerateInputsSimilarTo*. If we restrict the number of variables involved in predicates of invariants to n, for a program with p input parameters, there would be $O(\binom{p}{n}) = O(p^n)$ predicates. For a program with B branches, p parameters, with predicates restricted to 3 parameters, initially, the total number of predicates for all branches would be $O(B \times p^3)$. When actual test data are observed, most of these predicates are falsified and removed. On average, a fraction f of these predicates will stay valid per branch. If the average time of solving a constraint containing one predicate be C, and if we assume the time of solving a constraint involving more than one predicate be linear in the number of predicates, a rough estimation of the overhead of constraint solving for an iteration of the proposed method will be:

$$O(B \times p^3 \times f \times C) \tag{8}$$

For example, for a program with 100 branches, and 3 input parameters, where 0.1 of predicates stay valid in each branch likely invariant, with $C = 1$ ms, the overhead will be of order of 270 ms per iteration.

The factor with the most effect on the performance is the number of input parameters of the method under test. The proposed method will not scale well if the number of parameters in a method is too high. However, Martin, in his book Clean Code (Martin, 2009) states that functions should usually be restricted to at most 3 arguments, and functions with more than three arguments require very special justification. We analyzed libraries `apache commons math3` and `apache commons lang3` and measured the number of parameters of each method. These are the results (we show the results as p: m pairs, where p is the number of parameters, and m is the number of methods):

- `math3` which has 918 classes, 7191 methods, with an average number of parameters of 1.14: 0:2823, 1:2437, 2:1003, 3:456, 4:198, 5:147, 6:72, 7:7, 8:41, 9:2, 10:3, 11:1, 14:1.
- `lang3` which has 141 classes, 2745 methods, with an average number of parameters of 1.58: 0:508, 1:952, 2:706, 3:413, 4:106, 5:47, 6:9, 7:3, 9:1

These results indicate it is rare to encounter methods with more than 4 or 5 parameters.

If we apply the approach in Section 4.2, where generating similar input data is performed only when a parameter needs to be satisfied, we will get a better performance. This is because the number of constraint solvings is reduced.

The scheme we used in Section 4.2, where constraint solving is only performed when a method's parameters are needed to be satisfied, greatly decreases the number of required constraint solvings. Section 7 discusses some ideas to improve the performance of our method in future work.

5. Experiments and results

In this section, we show the results of evaluating the proposed method on multiple benchmark programs. The experiments are in two categories: experiments on the Python implementation of the proposed approach (Section 5.1), and experiments regarding the implementation of the proposed approach by extending EvoSuite (Section 5.2).

The first implementation of the proposed approach was in Python. For the python implementation, we wrote many elements, including the instrumentation code and search algorithms such as GA and PSO, from scratch. However, our python code only covers integer and boolean inputs, and some basic program structures. In order to be able to evaluate our approach on larger programs, and support more types, we decided to use EvoSuite (which is an excellent and industrial-level implementation of test data generation techniques), and change the relevant parts of EvoSuite to the proposed method.

5.1. Experiments regarding the python implementation

As the first evaluation of the proposed method, we implemented it in Python 3.6, and compared it with several test data generation methods. The goal was to cover all branches of benchmark programs. Evaluation metrics are the percentage of covered branches, percentage of full coverages in different runs (i.e., percentage of runs where 100% coverage is achieved), and the number of iterations needed to cover all branches. We used Microsoft Z3 for constraint solving.

5.1.1. Experimental setup

To conduct the comparison, we utilized several search-based algorithms. For GA, we used two implementations: regular single-goal genetic algorithm (Harman et al., 2012; Mishra et al., 2019) (shown as *ga:single*), and our version of multi-goal genetic algorithm which attempts to cover all branches simultaneously (shown as *ga:multi*). Our proposed method when employing the multi-goal genetic algorithm is shown as *ga-li*. Our implementation of PSO (shown as *pso*) corresponds to the method presented in Lv et al., 2018, Mao et al. (2012b); the proposed method when employing PSO is indicated by *pso-li*. Random search (shown as *random*), and simulated annealing (shown as *sa*) were also implemented. Furthermore, we used two implementations of ACO in the experiment: one based on the work of (Mao et al., 2015) (shown as *aco*), and the other one which is based on Sharifipour et al. (2018) (shown as *aco-es*). The method in Sharifipour et al. (2018) uses evolutionary strategy (ES) for local searches, in which a mutant of the current ant is replaced only if the mutant's fitness is better than the current ant's. At last, we implemented harmony search, shown as *harmony*, based on works (Mao, 2014; Sahoo et al., 2016). *ga:multi*, *ga-li*, and *pso-li* are multi-objective algorithms (target all goals simultaneously). Each algorithm was run 30 times on each program. To make the comparison fair, single solution methods, i.e., random and simulated annealing were performed on a population of 100 solutions.

The test data generation methods were run with the following parameters: the population size of 100, maximum number of iterations of 50, and range of integers from -2^{32} to 2^{32}. Eq. (4) was used for the fitness function, and the value of K in Fig. 3 was 1. For GA, we used a mutation probability of 0.01. For PSO, we updated velocities (denoted by $\mathbf{v}$) by formula $\mathbf{v}' = \mathbf{v} + c_1 r_1 (\mathbf{x}_{best} - \mathbf{x}) + c_2 r_2 (\mathbf{g} - \mathbf{x})$, where $c_1 = c_2 = 2$, r_1 and r_2 are random numbers in the range [0,1], $\mathbf{x}_{best}$ is the best known previous position for particle $\mathbf{x}$, and $\mathbf{g}$ is the global best position. The positions were updated by $\mathbf{x}' = \mathbf{x} + \mathbf{v}$. For ACO, the parameters were based on Mao et al. (2015). For Algorithm 6 of the proposed method, we used $q = 0.5$ and $p = 0.1$, which seemed to be suitable values based on our experiments on the benchmark programs. All the common parts (such as mutation and crossover operators, fitness functions, population size, mutation probabilities, etc) in different algorithms were exactly the same. We experimented the mentioned methods on benchmark programs of Table 1. Rows *LOC* and *Branches* show the lines of codes, and the number of branches of each program.

5.1.2. Results and discussion

Table 2 compares the results on metrics "branch coverage", "percentage of full coverages", and "the number of iterations" of running algorithms per benchmark program. Column *cov* shows the median branch coverage, and column *fcp* shows the percentage

Table 1
Benchmark programs in the first experiment.

	mid	gcd	triangle	dayOfWeek	numOfDays	quadratic_equation	point_circle	isValidDate	tcas	print_calendar	fisher
LOC	14	16	24	45	26	12	7	23	86	102	62
Branches	6	4	11	12	6	5	3	7	8	19	12

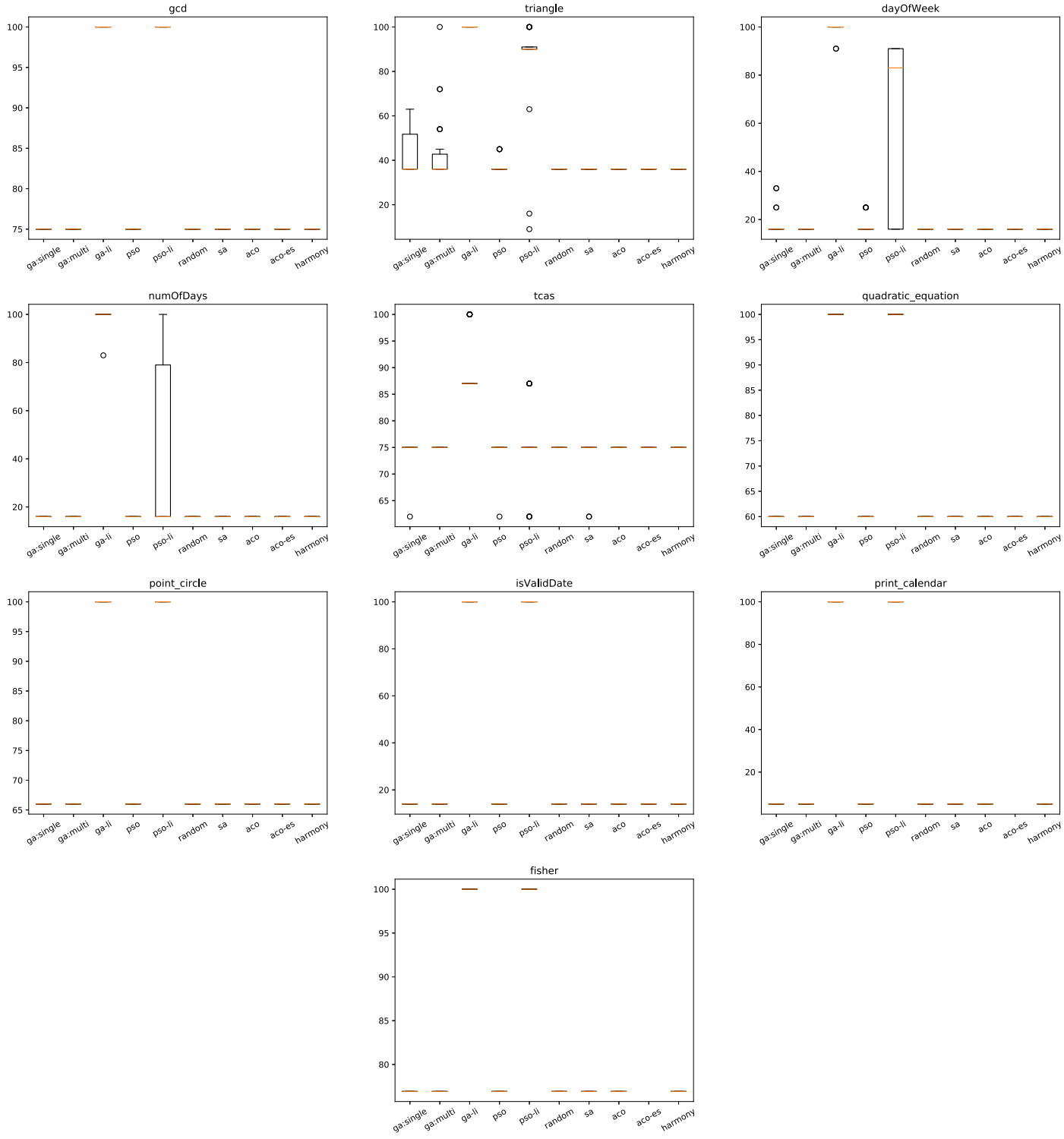

Fig. 11. Experiment results for the metric coverage (for a maximum of 50 iterations).

Table 2
Results of running different algorithms on benchmark programs.

	ga:single			ga:multi			ga-li			pso			pso-li			random			sa			aco			aco-es			harmony		
	cov	fcp	its	cov	fcp	its	cov	fcp	its	cov	fcp	its	cov	fcp	its	cov	fcp	its	cov	fcp	its	cov	fcp	its	cov	fcp	its	cov	fcp	its
mid	100	100	0	100	100	0	100	100	0	100	100	0	100	100	0	100	100	0	100	100	0	100	100	0	100	100	0	100	100	0
gcd	75	0	50	75	0	50	100	100	0	75	0	50	100	100	0	75	0	50	75	0	50	75	0	50	75	0	50	75	0	50
triangle	36	0	50	36	3.333	50	100	100	4	36	0	50	90	16.67	50	36	0	50	36	0	50	36	0	50	36	0	50	36	0	50
dayOfWeek	16	0	50	16	0	50	100	93.33	6	16	0	50	83	0	50	16	0	50	16	0	50	16	0	50	16	0	50	16	0	50
numOfDays	16	0	50	16	0	50	100	96.67	7	16	0	50	16	26.67	50	16	0	50	16	0	50	16	0	50	16	0	50	16	0	50
tcas	75	0	50	75	0	50	87	20	50	75	0	50	75	0	50	75	0	50	75	0	50	75	0	50	75	0	50	75	0	50
quadratic_equation	60	0	50	60	0	50	100	100	0	60	0	50	100	100	0	60	0	50	60	0	50	60	0	50	60	0	50	60	0	50
point_circle	66	0	50	66	0	50	100	100	0	66	0	50	100	100	0	66	0	50	66	0	50	66	0	50	66	0	50	66	0	50
isValidDate	14	0	50	14	0	50	100	100	1	14	0	50	100	100	1	14	0	50	14	0	50	14	0	50	14	0	50	14	0	50
print_calendar	5	0	50	5	0	50	100	100	0	5	0	50	100	100	0	5	0	50	5	0	50	5	0	50	5	0	50	5	0	50
fisher	76.92	0	50	76.92	0	50	100	100	2.5	76.92	0	50	100	100	1	76.92	0	50	76.92	0	50	76.92	0	50	76.92	0	50	76.92	0	50

Table 3
Results of running different algorithms on benchmark programs with the stopping criterion of time (20s).

	ga:single			ga:multi			ga-li			pso			pso-li			random			sa			aco			aco-es			harmony		
	cov	fcp	its	cov	fcp	its	cov	fcp	its	cov	fcp	its	cov	fcp	its	cov	fcp	its	cov	fcp	its	cov	fcp	its	cov	fcp	its	cov	fcp	its
gcd	100	86.67	253	75	0	1074	100	100	0	75	0	2277e	100	100	0	75	0	513	75	0	2953	75	0	42	75	0	41	75	0	677.5
triangle	36.36	0	1902	36.36	0	553	100	100	4.5	36.36	0	1741	90.91	40	14	36.36	0	3152	36.36	0	7643	36.36	0	86	36.36	0	90	36.36	0	2977
dayOfWeek	16.67	0	2981	16.67	0	484	100	91.67	4	16.67	0	2148	54.17	0	34.5	16.67	0	4578	16.67	0	5067	16.67	0	85	16.67	0	87	16.67	0	4077
numOfDays	16.67	0	2448	16.67	0	785	100	80	6.5	16.67	0	1845	16.67	10	2449	16.67	0	3414	16.67	0	9568	16.67	0	98	16.67	0	101	16.67	0	3590
tcas	75	0	163.5	75	0	438	87.5	3.333	24	75	0	508	75	0	4	75	0	709	75	0	1860	75	0	47	75	0	50	75	0	859.5
quadratic_equation	70	0	1100	80	0	1152	100	100	0	60	0	2409	100	100	0	60	0	1417	60	0	4412	60	0	57	60	0	62	60	0	1658
point_circle	66.67	0	1334	66.67	0	1144	100	100	0	66.67	0	869	100	100	0	66.67	0	1120	66.67	0	3980	66.67	0	39	66.67	0	38	66.67	0	1784
isValidDate	14.29	0	3289	14.29	0	789	100	100	1	14.29	0	2311	100	100	1	14.29	0	3655	14.29	0	10,240	14.29	0	91	14.29	0	90	14.29	0	4526
print_calendar	5	0	4886	5	0	322.5	100	100	0	5	0	5161	100	100	0	5	0	11,150	5	0	12,280	5	0	81	5	0	84	5	0	5616
fisher	76.92	0	722	76.92	0	437.5	100	100	2.5	76.92	0	589.5	100	100	1	76.92	0	1294	76.92	0	3022	76.92	0	32	76.92	0	35	76.92	0	1140

Table 4
The results of the ANOVA test on coverage means for *ga-li(a)* against other algorithms.

	ga:single(b)		ga:multi(c)		pso(d)		pso-li(e)		random(f)		sa(g)		aco(h)		aco-es(i)		harmony(j)	
	$m_a - m_b$	*p-value*	$m_a - m_c$	*p-value*	$m_a - m_d$	*p-value-*	$m_a - m_e$	*p-value*	$m_a - m_f$	*p-value*	$m_a - m_g$	*p-value*	$m_a - m_h$	*p-value*	$m_a - m_i$	*p-value*	$m_a - m_j$	*p-value*
gcd	25	0	25	0	25	0	0	undefined	25	0	25	0	25	0	25	0	25	0
triangle	57.7	1.89e-37	57.1	7.26e-29	62.8	3.76e-69	14.1	0.000512	64	0	64	0	64	0	64	0	64	0
dayOfWeek	81.7	1.32e-62	83.4	4.71e-84	82.2	1.68e-70	42.5	3.37e-08	83.4	4.71e-84	83.4	4.71e-84	83.4	4.71e-84	83.4	4.71e-84	83.4	4.71e-84
numOfDays	83.4	2.4e-76	83.4	2.4e-76	83.4	2.4e-76	61	2.65e-12	83.4	2.4e-76	83.4	2.4e-76	83.4	2.4e-76	83.4	2.4e-76	83.4	2.4e-76
tcas	15	1.54e-20	14.6	8.78e-22	15	1.54e-20	13.9	8.95e-12	14.6	8.78e-22	15.5	1.11e-19	14.6	8.78e-22	14.6	8.78e-22	14.6	8.78e-22
quadratic_equation	40	0	40	0	40	0	0	undefined	40	0	40	0	40	0	40	0	40	0
point_circle	34	0	34	0	34	0	0	undefined	34	0	34	0	34	0	34	0	34	0
isValidDate	86	0	86	0	86	0	0	undefined	86	0	86	0	86	0	86	0	86	0
print_calendar	95	0	95	0	95	0	0	undefined	95	0	95	0	95	0	95	0	95	0
fisher	23.1	2.29e-143	23.1	2.29e-143	23.1	2.29e-143	0	undefined	23.1	2.29e-143	23.1	2.29e-143	23.1	2.29e-143	23.1	2.29e-143	23.1	2.29e-143

Table 5
The results of the ANOVA test on iteration means for *ga-li(a)* against other algorithms.

	ga:single(b)		ga:multi(c)		pso(d)		pso-li(e)		random(f)		sa(g)		aco(h)		aco-es(i)		harmony(j)	
	$m_b - m_a$	*p-value*	$m_c - m_a$	*p-value*	$m_d - m_a$	*p-value*	$m_e - m_a$	*p-value*	$m_f - m_a$	*p-value*	$m_g - m_a$	*p-value*	$m_h - m_a$	*p-value*	$m_i - m_a$	*p-value*	$m_j - m_a$	*p-value*
gcd	50	0	50	0	50	0	0	undefined	50	0	50	0	50	0	50	0	50	0
triangle	46	1.64e-82	45.4	5.51e-59	46	1.64e-82	37.9	2.83e-16	46	1.64e-82	46	1.64e-82	46	1.64e-82	46	1.64e-82	46	1.64e-82
dayOfWeek	40.8	4.77e-27	40.8	4.77e-27	40.8	4.77e-27	40.8	4.77e-27	40.8	4.77e-27	40.8	4.77e-27	40.8	4.77e-27	40.8	4.77e-27	40.8	4.77e-27
numOfDays	41.2	1.23e-32	41.2	1.23e-32	41.2	1.23e-32	31.4	3.53e-12	41.2	1.23e-32	41.2	1.23e-32	41.2	1.23e-32	41.2	1.23e-32	41.2	1.23e-32
tcas	5.3	0.018	5.3	0.018	5.3	0.018	5.3	0.018	5.3	0.018	5.3	0.018	5.3	0.018	5.3	0.018	5.3	0.018
quadratic_equation	50	0	50	0	50	0	0	undefined	50	0	50	0	50	0	50	0	50	0
point_circle	50	0	50	0	50	0	0	undefined	50	0	50	0	50	0	50	0	50	0
isValidDate	48.6	2.26e-99	48.6	2.26e-99	48.6	2.26e-99	0.567	0.0475	48.6	2.26e-99	48.6	2.26e-99	48.6	2.26e-99	48.6	2.26e-99	48.6	2.26e-99
print_calendar	50	0	50	0	50	0	0	undefined	50	0	50	0	50	0	50	0	50	0
fisher	47.1	1.57e-28	47.1	1.57e-28	47.1	1.57e-28	-1.1	0.075	47.1	1.57e-28	47.1	1.57e-28	28.9	3.49e-24	28.9	3.49e-24	47.1	1.57e-28

of runs which achieved the 100% coverage in each algorithm. Column *its* indicates the median number of iterations (or generations) to achieve full coverage. Fig. 11 summarize the results of running the algorithms on benchmark programs in the form of box-plots which represent the distribution of results. We have omitted the diagrams for the *mid* program, because all the algorithms performed the same for it.

The results in Table 2 and Fig. 11 demonstrate the superiority of the proposed algorithm, especially when it is applied to GA. For all metrics, *ga-li* performs at least as well as all other methods, and in most cases outperforms them. These results also show that *pso-li* clearly performs better than *pso*. For other algorithms, in most cases *ga-single* and *ga-multi* perform slightly better than the rest of the algorithms. The weakest performances belong to *sa* (For example, it has the lowest coverage for *triangle*).

In order to show that the differences in results of *ga-li* and other algorithms are statistically significant, we performed the ANOVA (Analysis of Variance) test on average coverage and number of iterations of these algorithms. The null hypothesis H_0 is that there is no significant difference between the mean of the average coverage, and the mean of the number of iterations of *ga-li* against alternative algorithms. We choose the significance level α of 0.05. Tables 4 and 5 show the results of this analysis. In Table 4, columns $m_a - m_x$ indicate the difference of means of average coverages of *ga-li* (denoted by m_a) and algorithm *x*. *p-value* columns show the resulting *p-value* of the ANOVA test. In Table 5, columns $m_x - m_a$ indicate the difference of means of average iterations of algorithm *x* and *ga-li* (denoted by m_a).

For average coverages, we always have $m_a - m_x \geq 0$, for all algorithms *x*. Similarly, for average iterations, we always have $m_x - m_a \geq 0$. For coverage, the *p-value* for all cases where $m_a - m_x > 0$, is close to zero and far less than the significance level 0.05; hence, we have strong evidence to reject H_0. The same is correct for the number of iterations and cases where $m_x - m_a > 0$. The only entries where the results are equal to the results of *ga-li* belong to *pso-li*. Undefined *p-value*s happen when all the data in two groups are identical with variance 0. The results show that the differences reported in columns $m_a - m_x$ and $m_x - m_a$ in these two tables are statistically significant, and *ga-li* performs a lot better than the other methods. It is worth noting that in the current version of the proposed method, the time cost of a single iteration is more than the other algorithms, due to the overhead of constraint solving.

For the benchmark program *mid*, all methods reached to 100% coverage at the first iteration (the initial population phase), since satisfying the conditions of this program is very easy. For *gcd*, the branch with $a = 0 \ \vee \ b = 0$ is hard to cover due to the very low probability of satisfying this condition. The proposed method was able to cover this branch at the initial population phase, because of the use of predicates over constant values of the program text. None of the other methods were able to cover this branch in our experiments. For *triangle*, since the coverage of the "EQUILATERAL" branch (see Fig. 7) requires the establishment of the relation $(a = b = c) \wedge (a, b, c > 0)$, if the domain of each of these variables is $[-N, N]$, then the probability of selecting such a combination of random numbers is approximately equal to $\frac{N \times 1 \times 1}{(2N)^3} = \frac{1}{8N^2}$, which is a very small number for $N = 2^{32}$. The proposed method is able to discover relationships like $a = b$ or $b = c$ for related branches in the learning phase, which leads to adding high quality candidate solutions (with respect to branch(7)) to the population. Therefore, after a few iterations, it finds solutions satisfying $a = b = c$.

Due to the very limiting condition at the beginning of the programs *dayOfWeek* and *numOfDays* (which limits years, months and days to a very small range), the full coverage of these programs is hard, and the result was a low average coverage of the branches by most methods. Nevertheless, the use of predicates on program constants helps *ga-li* to reach to an average of nearly 100% coverage for these programs. For *quadratic_equation, point_circle, print_calendar, isValidDate*, and *fisher*, both *ga-li* and *pso-li* perform equally well, and far better than the other methods, by learning the relationships among their input variables, and using constant values of the program sources (e.g., the limits imposed on *month* or *day* variables in *isValidDate*). *tcas* receives 12 input variables, and finding the right relationships among these variables to reach a full coverage is very difficult; this explains why *ga-li* only reached full coverage in 20% of times. However, it still outperforms all other algorithms in terms of all evaluation metrics for this benchmark program.

We conducted another experiment to measure the efficiency of the proposed method. The results are shown in Table 3. In this experiment, we set the stopping condition a runtime of 20 s. The performance of *ga-li* is similar to the case where the stopping condition was iteration count of 50. In this case too, the only benchmark

Table 6
Java benchmark programs.

	LOC	*Branches*
Triangle	40	35
LineRectangle	96	75
ComputeTax	159	49
PrintCalendar	183	99
TCAS (jpf)	233	86
Fisher (evo)	74	17
Bessj (evo)	133	29
days_number	230	107
Gammq (evo)	92	27
WBS (jpf)	262	92
MathSin (jpf)	474	55
Rational (jpf)	281	79
JulianDate	102	15
BMI	18	9
Expint (evo)	88	31
SmallPrimes (math3)	189	34
PollardRho (math3)	165	22
ArithmeticUtils (math3)	908	158
BitsStreamGenerator (math3)	271	35
EmpiricalDistribution (math3)	102	88
HaltonSequenceGenerator (math3)	184	19
FiniteDifferencesDifferentiator (math3)	385	54
BicubicInterpolatingFunction (math3)	326	49
TricubicInterpolatingFunction (math3)	509	92
Complex (math3)	1301	196
DfpDec (math3)	369	138
BetaDistribution(math3)	407	57
Arc (math3)	133	21
Primes (math3)	130	26
BesselJ (math3)	649	137
Fraction (lang3)	925	202
NumberUtils (lang3)	1826	440
BooleanUtils (lang3)	1102	269
RandomUtils (lang3)	236	32
Beta (math3)	514	96
Gamma (math3)	721	103
Erf (math3)	244	29
FastMath (math3)	4297	894
CombinatoricsUtils (math3)	463	99
Precision (math3)	609	146
IntMath (guava)	729	253
LongMath (guava)	1208	381
PostCodeValidator (evo)	161	12
HardConstraints (evo)	114	35
WordUtils (lang3)	744	103
TestZ3 (jpf)	220	53
TP1 (evo)	1845	1071
TP118 (evo)	1604	981
DoubleMath (guava)	529	137
Remainder (evo)	68	25
Quantiles (guava)	686	109

Table 7
The results of experiments on the integration of Evosuite with the proposed method, with the stopping condition of 30 iterations.

	EVO		EVO+LI		
	Avg Fitness	*Avg Uncovered*	*Avg Fitness*	*Avg Uncovered*	*p-value*
Triangle	2.698	2.8	**0.82**	**1.2**	0.022
LineRectangle	30.52	31.5	**17.75**	**21.4**	5e-05
ComputeTax	34.9	34.9	**27.2**	**27.5**	0.00075
PrintCalendar	3.209	5.4	**2.227**	**4.05**	0.022
TCAS	40.35	45	**27.78**	**32.3**	0.0048
Fisher	**0.3167**	**0.5**	0.5167	0.9	0.37
Bessj	**2.8**	**2.8**	3.1	3.1	0.2
days_number	99.02	99.07	**33.53**	**38**	2.7e-08
Gammq	3.851	4.467	**3.091**	**3.8**	0.019
WBS	18.6	28	**18.55**	28	0.33
MathSin	7.801	9.5	**5.285**	**8.2**	0.0025
Rational	25.37	27	**19.65**	**22.4**	0.011
JulianDate	**1.908**	**2**	2.366	2.5	0.25
BMI	1.475	**1.75**	**1.45**	1.9	0.96
Expint	5.768	6.333	**4.511**	**5.171**	0.035
SmallPrimes	20.64	20.65	**16.76**	**17.02**	0.041
PollardRho	**4.045**	**4.909**	4.518	5.5	0.51
ArithmeticUtils	**51.36**	**60.15**	58.07	65.9	0.14
BitsStreamGenerator	5.483	6.071	**4.695**	**5.333**	0.047
EmpiricalDistribution	**48.56**	47	49.06	**46.5**	0.93
HaltonSequenceGenerator	7.865	8.45	**4.342**	**5.722**	0.032
FiniteDifferencesDifferentiator	46.03	42.3	**45.95**	**42.2**	0.91
BicubicInterpolatingFunction	40.5	39.83	**40.19**	**39.4**	0.85
TricubicInterpolatingFunction	76.07	76	**71.92**	**72.5**	0.037
Complex	57.85	78.6	**50.89**	**70.9**	0.049
DfpDec	**138**	**135**	139.2	136.2	0.037
BetaDistribution	**16.78**	**18.5**	21.93	24	0.11
Arc	13.73	13.9	**10.03**	**10.9**	0.011
Primes	16.16	15.8	**11.56**	**12.27**	0.039
BesselJ	**77.81**	**82.4**	83.51	89.5	0.72
Fraction	79.96	**93.44**	**79.64**	93.67	0.95
NumberUtils	**397.2**	**377.5**	404.9	383.4	0.46
BooleanUtils	**243.5**	**231.8**	244.2	232.2	0.9
RandomUtils	14.99	16.35	**8.602**	**9.52**	0.012
Beta	20.18	25.5	**10.48**	**16.8**	0.0017
Gamma	22.48	29.89	**4.867**	**9**	0.0059
Erf	4.533	6.12	**1.856**	**2.8**	0.023
FastMath	218.3	246.4	**202.6**	**231.2**	0.72
CombinatoricsUtils	47.54	49	**33.2**	**37.9**	0.00011
Precision	**39.32**	49.3	39.56	49.3	0.92
IntMath	81.3	99.2	**70.91**	**90.2**	0.0077
LongMath	179.2	193.7	**137.7**	**157.1**	7e-06
PostCodeValidator	0.381	0.6905	**0.1486**	**0.2973**	0.035
HardConstraints	18.05	19.5	**13.99**	**15.2**	6.4e-06
WordUtils	1.876	3.455	**1.152**	**2.182**	0.025
TestZ3	13.49	16.4	**11.49**	**13.9**	0.029
TP1	373.7	419.2	**344.1**	**399.6**	0.024
TP118	311.3	351.3	**285.9**	**331.6**	2.1e-05
DoubleMath	60.66	71.9	**60.2**	**71**	0.93
Remainder	7.846	8.3	**6.766**	**7.425**	0.36
Quantiles	16.33	21.15	**15.74**	**20.42**	0.85

program which *ga-li* didn't reach to 100% coverage, in all runs was *tcas*, which is because it has 12 parameters, and the time required for its constraint solving exceeded the budget of 20 s.

In most cases, the proposed method performed better when applied to GA compared to PSO. What matters most in GA, is the genetic information contained in the data. When using constraint solving, the generated solutions are themselves enough to guide GA to the right direction for searching. On the other hand, in swarm intelligence algorithms, such as PSO, the quality of an individual depends on properties in addition to the goodness of their values. For example, in PSO, not only an individual has a position, but also it has a value for the property velocity as well. For newly generated individuals by using constraint solving, we set the values of these individual-specific properties to the values of the properties of the best currently found solution. This approach may be insufficient, and it explains why in most cases GA performs better.

The results show that the proposed method leads to significant enhancement in the ability of search-based algorithms (such as GA and PSO) to cover different parts of programs. GA when improved with the proposed method was able to reach 100% coverage more than any of the other methods. Overally, in all of the experiments, when search-based methods were combined with the proposed method, they performed better compared to the cases where they were used alone (in terms of the number of iterations, percentage of full coverages and total coverage).

5.2. Experiments regarding the integration of the proposed method with EvoSuite

We implemented a prototype of the proposed method in Java, by extending EvoSuite. We compared the EvoSuite's implementation of Whole Suite (refer to Section 3), denoted by **EVO**, with the integration of our method with the EvoSuite's Whole Suite, de-

Table 8
The results of experiments on the integration of Evosuite with the proposed method, with the stopping condition of 20 s.

	EVO		EVO+LI		
	Avg Fitness	*Avg Uncovered*	*Avg Fitness*	*Avg Uncovered*	*p-value*
Triangle	**1.098**	**1.2**	2.218	2.7	0.062
LineRectangle	8.953	10.2	**4.39**	**6.1**	0.00028
ComputeTax	24.3	24.4	**21.15**	**21.2**	0.24
PrintCalendar	**0.55**	**1.1**	7.404	11.5	4.1e-08
TCAS	**26.12**	**30.8**	31.79	37.3	0.068
Fisher	**0**	**0**	0.2333	0.4	0.089
Bessj	**2**	**2**	3.25	3.3	0.0062
days_number	**100.2**	**100.2**	101	101.1	0.36
Gammq	**3.949**	**4.6**	6.302	6.6	0.00067
WBS	**19.58**	**29.2**	21.56	31.8	0.12
MathSin	**7.57**	**9.6**	17.57	19.7	2.6e-06
Rational	**23.35**	**25**	35.13	35.6	6.9e-05
JulianDate	**0**	**0**	4.049	4.4	3.2e-06
BMI	**2.067**	**2.6**	3.567	4	0.073
Expint	**5.046**	**5.5**	11.8	13	0.00024
SmallPrimes	**11.17**	**11.8**	18.48	18.8	0.0013
PollardRho	**3.286**	**4.1**	7.648	8.2	4.8e-05
ArithmeticUtils	**69.14**	**74.3**	87.61	89.7	0.00036
BitsStreamGenerator	**7.237**	**8.1**	9.4	10.4	0.065
EmpiricalDistribution	**40.08**	**39**	51.88	48.9	0.14
HaltonSequenceGenerator	9.601	9.7	**8.091**	**8.7**	0.62
FiniteDifferencesDifferentiator	**43.73**	**39.8**	45.97	42.1	0.014
BicubicInterpolatingFunction	**39.02**	**38.6**	44.8	43	0.069
TricubicInterpolatingFunction	**83.76**	**82.1**	83.81	82.3	0.99
Complex	**54.02**	**74.8**	71.32	87.9	0.0054
DfpDec	**136.6**	**133.6**	139.7	136.7	3.1e-05
BetaDistribution	19.58	21.4	**17.23**	**19.4**	0.6
Arc	**10.38**	**10.5**	12.58	12.7	0.36
Primes	**8.95**	**9.5**	22.09	21.1	0.00017
BesselJ	79.6	84.5	**76.98**	**82.7**	0.89
Fraction	106.3	111.3	**100**	**109.7**	0.68
NumberUtils	**414.2**	**390.5**	427.3	399	0.17
BooleanUtils	**178**	**184.3**	209.9	206.7	0.064
RandomUtils	**3.58**	**4.9**	20.62	19.7	4.2e-09
Beta	**11.84**	**16.2**	17.18	22.3	0.025
Gamma	**3.26**	**5.5**	16.12	21.9	4.8e-10
Erf	**0.4167**	**0.7**	6.167	7.4	1.8e-08
FastMath	**350.3**	**397.3**	591	596	1.3e-10
CombinatoricsUtils	**34.66**	**36.7**	53.23	55.5	0.001
Precision	**55.21**	**65**	60.55	68.9	0.22
IntMath	**155.7**	**164.4**	233.6	223.5	1.7e-08
LongMath	**219.9**	**239.1**	306.9	301.7	1.1e-06
PostCodeValidator	**1.4**	**1.7**	6.2	6.8	1.3e-05
HardConstraints	**25.37**	**25.6**	29.88	29	0.00086
WordUtils	**11.98**	**19**	53.61	55.4	7.9e-08
TestZ3	**21.41**	**23.8**	36.49	34	7.6e-05
TP1	416.8	452.6	**382.3**	**434**	0.019
TP118	341.1	**376**	**336.4**	381	0.64
DoubleMath	**71**	**78.8**	114.2	108.6	6.5e-07
Remainder	**7.412**	**7.8**	9.26	10	0.32
Quantiles	**30.61**	**34**	62.37	62	9.3e-05

noted by **EVO+LI**. The metrics were *Fitness* (which is the fitness value of a test suite that EvoSuite calculates for the branch coverage criterion) and the number of *Uncovered* branches of the class under test. This section presents the setup and the results of the experiments on 51 Java benchmarks.

5.2.1. Experimental setup

We ran **EVO** and **EVO+LI** on 51 Java benchmarks, including some classes of `apache commons math3`, `apache commons lang3`, and `google guava`. Other benchmarks include some classes from EvoSuite (evo) or Java PathFinder (jpf)'s test programs, as well as some standard test programs used in the test data generation literature. These classes are shown in Table 6. Columns *LOC* and *Branches* show the lines of codes, and the number of branches of each class. All the classes have methods which contain numeric or string parameters. Some of these classes are large; for example, `FastMath.java` contains 4297 lines of code.

The setup for **EVO** and **EVO+LI** was as follows: a search budget (the EvoSuite equivalent of the maximum number of iterations) of 30, and population size of 10, where each chromosome had a maximum number of 40 statements. The default configurations of EvoSuite are a search budget of 60, with a population size of 50. We have chosen a smaller number of generations and a smaller population size to emphasize the improvement that the proposed method offers under limited circumstances. The difference in effectiveness is more pronounced when we are using fewer individuals in the population. The probability of using likely invariants for satisfying a supported parameter (LI_PROBABILITY in Fig. 9) was set to 0.7. The supported parameter types were numeric types (int, long, short, float, double), boolean, and string. All other types used the

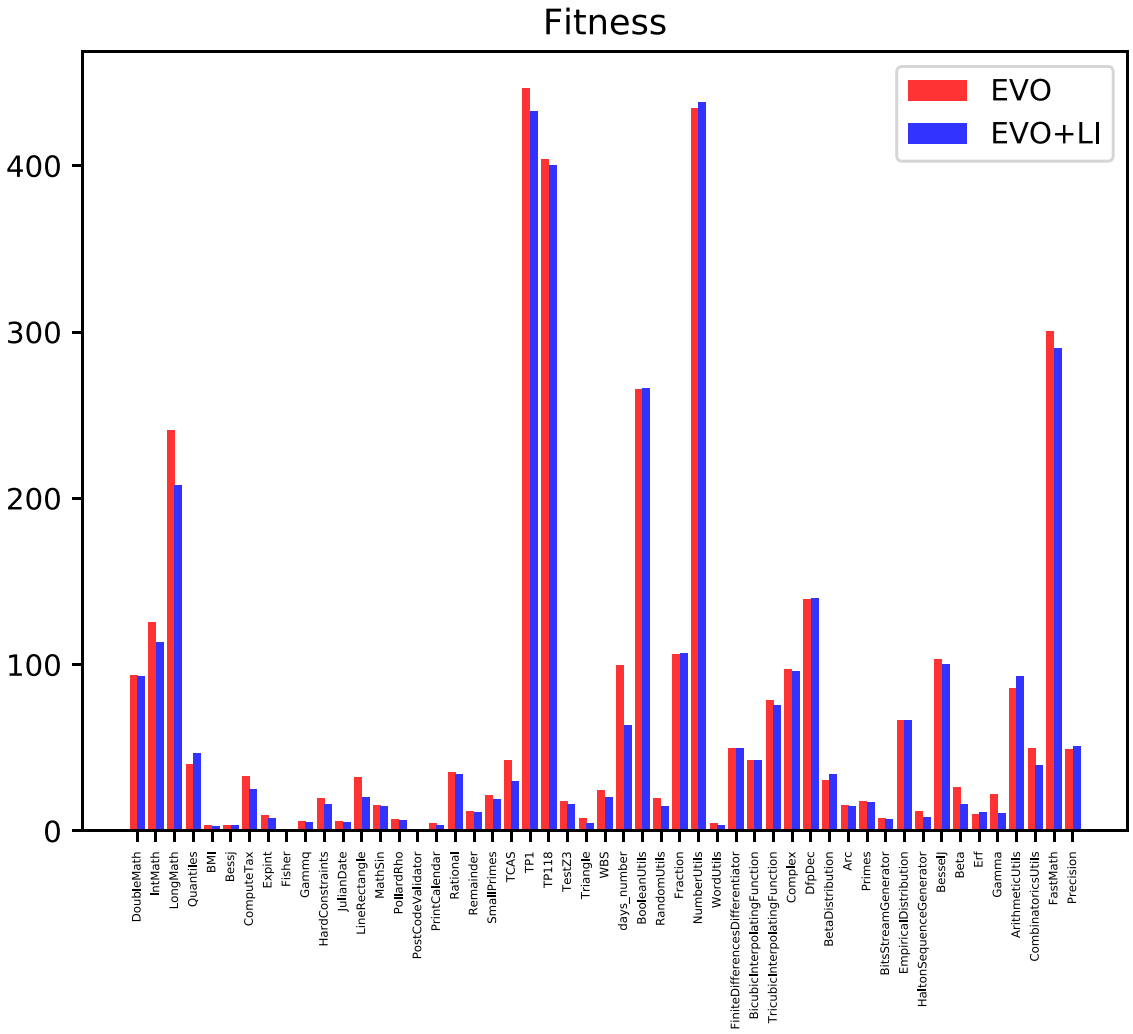

Fig. 12. Experiment results for fitness values (0 means complete coverage): EvoSuite's algorithm (EVO) vs EvoSuite's algorithm + the proposed method (EVO+LI).

default mechanism of EvoSuite. The test criterion was branch coverage. Each algorithm was run for each class at least 10 times.

5.2.2. Results

Table 7 shows the results of the experiment. Columns *Avg fitness* and *Avg Uncovered* indicate the average fitness and the number of uncovered branches of the class under test. *p-value* shows the *p-value* of the ANOVA test for the *Fitness* metric. Fig. 12 compares the performance of **EVO** vs **EVO+LI** for the metric *Fitness*.

In 76.5% of cases, **EVO+LI** gained a better average fitness for the same number of iterations. In these cases, the average decrease of fitness was 8.27. Using the proposed approach, the reduction in the number of iterations needed to cover different branches is more evident with benchmark programs having branches with conditions more depended on the numeric or string input parameters (such as ComputeTax, TCAS, days_numbers, LineRectangle, CombinatoricsUtils). When the proposed method did not have a positive effect on the EvoSuite's algorithm, it's usually because the conditions were not dependent on numeric or string parameters, or if they were, satisfying those conditions was easy enough for both algorithms.

We conducted another experiment to compare the efficiency of **EVO** and **EVO+LI**. In this experiment, the stopping condition was a runtime of 20 s (instead of 30 generations). Table 8 shows the results of this experiment. In 84.3% of cases, the EvoSuite's original algorithm performs more efficiently. This means that although constraint solving decreases the number of required iterations, each iteration of our implementation is slower. Therefore, in terms of efficiency (doing the same work in less time), the EvoSuite's original algorithm performs much better.

In order to measure the effectiveness of using likely invariants for the generation of the initial population, we repeated the experiments, but, this time we set the SEARCH_BUDGET configuration of EvoSuite to 1. This configuration causes the algorithm to not run the evolution phase of the search algorithms, or in other words, only the initialization step is executed. EvoSuite uses both constant and dynamic seeding to initialize the population of its algorithm. Table 9 shows the results of executing **EVO** and **EVO+LI** with SEARCH_BUDGET=1. The average fitness was better in 62.7% of benchmark programs for **EVO+LI**. Therefore, using likely invariants for the generation of initial population, leads to an improvement in the average fitness. As previously mentioned, for SEARCH_BUDGET = 30, in 76.5% of cases **EVO+LI** gained a better average fitness. Comparing these percentages, we observe that the more iterations the search algorithm performs, the results get better. This is due to the fact that by going through more iterations, and observing more input data, the algorithm can achieve a better approximation of the true invariants of branches, which leads to generating input values with higher fitness.

The results show our changes made an improvement in the effectiveness of the EvoSuite's algorithm (better coverage for the same number of iterations) for programs with numeric or string

Table 9
Comparison of fitness of the initial population (SEARCH_BUDGET = 1) without using versus with using likely invariants.

	EVO		EVO+LI		
	Avg Fitness	*Avg Uncovered*	*Avg Fitness*	*Avg Uncovered*	*p-value*
Triangle	12.08	13	**8.671**	**10.2**	0.00041
LineRectangle	34.38	35.4	**23.13**	**26.2**	0.00016
ComputeTax	30.8	30.8	**22.88**	**22.9**	0.00051
PrintCalendar	8	11.8	**5.756**	**9.5**	0.0043
TCAS	44.48	47.7	**31.81**	**35.8**	5.3e-05
Fisher	0.6583	1	**0.525**	**0.8**	0.56
Bessj	3.758	3.9	**3.1**	**3.25**	0.011
days_number	99.97	100	**76.69**	**78.32**	2.3e-05
Gammq	7.507	7.767	**6.619**	**6.929**	0.074
WBS	30.43	40.3	**21.98**	**32.3**	0.00018
MathSin	**18.8**	**20.7**	19.09	20.85	0.84
Rational	**40.09**	**39.25**	40.95	39.65	0.59
JulianDate	7.676	7.55	**6.749**	**6.8**	0.3
BMI	5.383	5.4	**4.417**	**4.6**	0.1
Expint	17.7	18	**13.37**	**14**	0.0051
SmallPrimes	22.54	21.6	**21.19**	**20.73**	0.45
PollardRho	9.825	9.7	**7.664**	**8.3**	0.042
ArithmeticUtils	120.1	113.1	**110.7**	**105.8**	0.024
BitsStreamGenerator	**21.64**	20.2	21.99	**20**	0.85
EmpiricalDistribution	**84.1**	**72.4**	84.33	72.7	0.94
HaltonSequenceGenerator	15.16	13.65	**14.68**	**13.25**	0.75
FiniteDifferencesDifferentiator	**51.01**	**47.1**	51.25	47.4	0.78
BicubicInterpolatingFunction	46.56	43.71	**46.23**	**43.44**	0.7
TricubicInterpolatingFunction	90.22	86.35	**88.09**	**84.8**	0.14
Complex	**116.9**	117.4	118.6	**117**	0.7
DfpDec	140.3	137.3	140.3	137.3	1
BetaDistribution	**37.53**	**36.8**	39.8	37.8	0.39
Arc	**15.79**	**15.4**	17.52	16.85	0.054
Primes	**21.04**	**19.6**	21.71	20.3	0.74
BesselJ	115.9	115.9	**108.1**	**108.5**	0.18
Fraction	153.9	146.6	**147.6**	**142.1**	0.49
NumberUtils	**452.9**	**414.6**	454.7	414.9	0.7
BooleanUtils	**287.6**	259.1	287.7	**259**	0.98
RandomUtils	**28.37**	**25.4**	29.93	26.5	0.26
Beta	32.1	33.6	**21.92**	**26.9**	0.00023
Erf	**23.38**	**21.9**	25.28	23.2	0.28
FastMath	**579.7**	**585.9**	597.5	600.2	0.16
CombinatoricsUtils	51.95	52.6	**46.08**	**49.5**	0.015
Gamma	21.42	26.8	**15.88**	**21.7**	0.023
Precision	**58.41**	**67.1**	61.58	70.7	0.27
IntMath	147	154.2	**134.6**	**144.4**	0.024
LongMath	302.7	296.3	**277.9**	**274.1**	0.039
PostCodeValidator	**0.7**	**1.1**	0.85	1.2	0.43
HardConstraints	20.9	21.6	**17.74**	**18.8**	0.00037
WordUtils	7.597	12.6	**5.989**	**10.3**	0.036
TestZ3	19.37	21.4	**17.39**	**19.67**	0.046
TP1	482.4	**515.1**	**477.1**	522.6	0.66
TP118	**434.7**	**464.3**	437.9	474.2	0.82
Quantiles	**63.17**	**62.7**	75.6	73.1	0.0096
DoubleMath	109.8	105.3	**109.1**	**104.3**	0.87
Remainder	19.68	**19.4**	**19.6**	19.45	0.96

inputs, and branches with conditions relying on those inputs. The proposed method was able to decrease the fitness value (and the number of uncovered goals) for these programs. On the other hand, the efficiency of the EvoSuite's default algorithm is better.

6. Threats to validity

A potential source of bias in the results comes from the inherent random behavior of the search-based methods. To overcome this problem, one approach is to apply these methods reasonably enough times on benchmark programs. Therefore, we repeated our experiments at least 10 times. Furthermore, to show that the differences in results for different methods are statistically significant, we used an ANOVA test with the significance level of 0.05. Another threat is related to the parameters used in the algorithms. We tried to make the comparisons as fair as possible by keeping the common parts (such as parameters, fitness functions, and common subroutines) exactly the same for all the implemented algorithms.

In this work, we are far away from using the full potential of likely invariants in discovering the relationships between program parameters. There exists a great number of useful invariants over numeric and string types that can happen, but we did not utilize. Moreover, although we have discussed invariants for other types in Section 4.3, we have not covered non-numeric or non-string types in our experiments. Unfortunately, generalization to other types (both in theory and implementation) is not trivial, and this problem needs to be addressed in future works.

As shown in Section 4.4.3, the proposed method may not scale well when the number of input parameters increase. However, as indicated in Section 4.4.3, it is rare for functions to have a large number of parameters. While the results show that the proposed

method has a positive impact on the effectiveness of each iteration (coverages in individual iterations), but due to constraint solving, each iteration of the algorithm takes much longer than a typical iteration of a search-based algorithm. Efficiency is one of the issues that we plan to actively work in our future researches.

7. Conclusion and future work

In this paper, we proposed an approach to improve search-based test data generation methods. The method is based on learning the existing relationships between program inputs and parts of the program these inputs cover. In order to do so, we introduced the concepts branch likely invariant and path likely invariant. We also showed how to use potential existing relationships among program input variables and constants to improve the fitness of first generations. Since utilizing the branch likely invariants in the proposed method increases the number of individuals (e.g., chromosomes in case of the genetic algorithm) with higher fitness, the likelihood of achieving the final goal in fewer iterations becomes much higher.

In the following subsections, we discuss some ideas for future work.

7.1. Implementing the support for other types

The focus in this work was on numeric and string types, and the implementations were limited to parameters with numeric, boolean or string types. However, in this section, we provide a list of invariants that need to be supported in order to be able to generalize the proposed method to some other types. Some of the following invariants were introduced in Ernst and Notkin (2000) (especially invariants concerning arrays). In order to be able to incorporate these invariants in the proposed method, one needs to implement code to check the validity of given arrays or objects against these predicates, as well as implementing code to solve constraints containing these invariants.

For sequences (arrays or lists) A and B, we can consider these invariants ($|A|$ denotes the length of A):

- A is null, or A is empty ($|A| = 0$)
- $|A| = |B|$
- A and B are equal: $|A| = |B| \wedge \forall i \in 0.|A|.A[i] = B[i]$

If A is a sequence of comparable items, we can have invariants about the sortedness of A:

- Ascending: $\forall i, j \in 0.|A|.i \leq j \Rightarrow A[i] \leq A[j]$
- Descending: $\forall i, j \in 0.|A|.i \leq j \Rightarrow A[i] \geq A[j]$

If x is the same type as the elements of A, we can have $x \in A$, or $x \notin A$. If A and B is a sequence of numeric types, there can be invariants such as $A = \alpha B + \beta$, or $\forall x \in A.x = 0$.

For any object obj of type T, we can consider these invariants:

- $obj = null$
- $obj \neq null$

By the same technique as dynamic seeding (Rojas et al., 2016), we can store a pool $P(T)$ of objects of type T. We can use these objects for constraints, too. For example, we can have:

- $obj = t$ for some $t \in P(T)$

If obj has a public property p of type X, and if another parameter x of type X is available, we can compare these values:

- $obj.p = x$

If obj has a public property i of numeric, array or string type, we can consider $obj.i$ as another independent parameter, and apply all the aforementioned invariants (corresponding to the type of i) on $obj.i$ as well.

Z3 supports theory of arrays and select-store axioms (McCarthy, 1959). Select(A, i) is equivalent to $A[i]$ and $Store(A, i, v)$ is equivalent to $A[i] = v$ in programming languages.

7.2. Performance of the proposed method

In future research, the performance of the proposed method should be considered in more details. Although the proposed method decreases the number of required iterations to cover specified goals, but due to the need for constraint solving in each iteration, the overhead is more than common methods of search based test data generation. Therefore, it is necessary to minimize the number and complexity of constraints (when generating similar program inputs to the currently best found ones) to maximize the method's performance. The performance and efficiency of the proposed method can be increased by a number of ideas:

- Restricting constraint solving by the use of a control dependency graph (Panichella et al., 2017): we can use a control dependency graph, to only solve constraints for uncovered goals immediately reachable from covered goals.
- Smarter use of predicates, by removing predicates that are logical consequences of other predicates,
- Caching: we can use a caching scheme for constraints, to avoid solving the same constraints over and over.
- Limiting the initial domains of constraint variables, and progressively widening them.
- For simple constraints, the use of constraint solvers can be avoided, by randomly selecting values from the domains of variables.
- Dynamic seeding: we can use values obtained in dynamic seeding and incorporate them in constraints. For example, if value 100 is encountered in the execution of the program, we can make constraints such as $x > 100$ (if x is an integer parameter) during runtime.

Declaration of Competing Interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

M. Nosrati: Conceptualization, Software, Formal analysis, Investigation, Writing - original draft, Writing - review & editing, Visualization. **H. Haghighi:** Conceptualization, Writing - review & editing, Supervision, Project administration. **M. Vahidi Asl:** Conceptualization, Writing - review & editing.

References

Anand, S., 2012. Techniques to Facilitate Symbolic Execution of Real-World Programs. Georgia Institute of Technology.
Arcuri, A., 2018. Test suite generation with the many independent objective (MIO) algorithm. Inf. Softw. Technol. 104, 195–206.
Ayari, K., Bouktif, S., Antoniol, G., 2007. Automatic mutation test input data generation via ant colony. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation. ACM, pp. 1074–1081.
Baars, A., Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Tonella, P., Vos, T., 2011. Symbolic search-based testing. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, pp. 53–62.
Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I., 2018. A survey of symbolic execution techniques. ACM Comput. Surv. (CSUR) 51 (3), 50.
De Moura, L., Bjørner, N., 2008. Z3: an efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 337–340.

DeMilli, R., Offutt, A.J., 1991. Constraint-based automatic test data generation. IEEE Trans. Softw. Eng. 17 (9), 900–910.
Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D., 2001. Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Softw. Eng. 27 (2), 99–123.
Ernst, M.D., Notkin, D., 2000. Dynamically Discovering Likely Program Invariants. University of Washington.
Ferrer, J., Chicano, F., Alba, E., 2012. Evolutionary algorithms for the multi-objective test data generation problem. Software 42 (11), 1331–1362.
Fraser, G., Arcuri, A., 2011. Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ACM, pp. 416–419.
Fraser, G., Arcuri, A., 2012. The seed is strong: seeding strategies in search-based software testing. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE, pp. 121–130.
Fraser, G., Arcuri, A., 2013. Whole test suite generation. IEEE Trans. Softw. Eng. 39 (2), 276–291.
Fraser, G., Arcuri, A., 2014. A large-scale evaluation of automated unit test generation using evosuite. ACM Trans. Softw. Eng.Method. (TOSEM) 24 (2), 8.
Fraser, G., Arcuri, A., 2015. 1600 Faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. Empir. Softw. Eng. 20 (3), 611–639.
Girgis, M.R., 2005. Automatic test data generation for data flow testing using a genetic algorithm. J. Univers. Comput. Sci. 11 (6), 898–915.
Harman, M., McMinn, P., 2007. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis. ACM, pp. 73–83.
Harman, M., McMinn, P., De Souza, J.T., Yoo, S., 2012. Search based software engineering: techniques, taxonomy, tutorial. In: Empirical Software Engineering and Verification. Springer, pp. 1–59.
King, J.C., 1976. Symbolic execution and program testing. Commun. ACM 19 (7), 385–394.
Korel, B., 1990. Automated software test data generation. IEEE Trans. Softw. Eng. 16 (8), 870–879.
Lakhotia, K., Harman, M., McMinn, P., 2007. A multi-objective approach to search-based test data generation. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation. ACM, pp. 1098–1105.
Li, H., Lam, C.P., 2004. Software test data generation using ant colony optimization. In: International Conference on Computational Intelligence, pp. 1–4.
Li, K., Zhang, Z., Liu, W., 2009. Automatic test data generation based on ant colony optimization. In: Natural Computation, 2009. ICNC'09. Fifth International Conference on, vol. 6. IEEE, pp. 216–220.
Lv, X.-W., Huang, S., Hui, Z.-W., Ji, H.-J., 2018. Test cases generation for multiple paths based on PSO algorithm with metamorphic relations. IET Softw 12 (4), 306–317.
Majumdar, R., Sen, K., 2007. Hybrid concolic testing. 29th International Conference on Software Engineering (ICSE'07). IEEE, pp. 416–426.
Malburg, J., Fraser, G., 2011. Combining search-based and constraint-based testing. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, pp. 436–439.
Manaseer, S., Manasir, W., Alshraideh, M., Hashish, N.A., Adwan, O., 2015. Automatic test data generation for java card applications using genetic algorithm. J. Softw. Eng. Appl. 8 (12), 603.
Mao, C., 2014. Harmony search-based test data generation for branch coverage in software structural testing. Neural Comput. Appl. 25 (1), 199–216.
Mao, C., Xiao, L., Yu, X., Chen, J., 2015. Adapting ant colony optimization to generate test data for software structural testing. Swarm Evol. Comput. 20, 23–36.
Mao, C., Yu, X., Chen, J., 2012. Swarm intelligence-based test data generation for structural testing. In: Computer and Information Science (ICIS), 2012 IEEE/ACIS 11th International Conference on. IEEE, pp. 623–628.
Mao, C., Yu, X., Chen, J., Chen, J., 2012. Generating test data for structural testing based on ant colony optimization. In: Quality Software (QSIC), 2012 12th International Conference on. IEEE, pp. 98–101.
Mariani, L., Pezze, M., Zuddas, D., 2015. Recent advances in automatic black-box testing. In: Advances in Computers, vol. 99. Elsevier, pp. 157–193.
Martin, R.C., 2009. Clean Code: A Handbook of Agile Software craftsmanship. Pearson Education.
McCarthy, J., 1959. A basis for a mathematical theory of computation. In: Studies in Logic and the Foundations of Mathematics, vol. 26. Elsevier, pp. 33–70.
McMinn, P., 2004. Search-based software test data generation: a survey. Softw. Test. Verif. Reliab. 14 (2), 105–156.
Michael, C.C., McGraw, G.E., Schatz, M.A., Walton, C.C., 1997. Genetic algorithms for dynamic test data generation. In: Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference. IEEE, pp. 307–308.
Mishra, D.B., Mishra, R., Acharya, A.A., Das, K.N., 2019. Test data generation for mutation testing using genetic algorithm. In: Soft Computing for Problem Solving. Springer, pp. 857–867.
Nayak, N., Mohapatra, D.P., 2010. Automatic test data generation for data flow testing using particle swarm optimization. Contemp. Comput. 1–12.
Offutt, J., Ammann, P., 2008. Introduction to Software Testing. Cambridge University Press Cambridge.
Panichella, A., Kifetew, F.M., Tonella, P., 2015. Reformulating branch coverage as a many-objective optimization problem. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp. 1–10.
Panichella, A., Kifetew, F.M., Tonella, P., 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Trans. Softw. Eng. 44 (2), 122–158.
Pargas, R.P., Harrold, M.J., Peck, R.R., 1999. Test-data generation using genetic algorithms. Softw. Test. Verif. Reliab. 9 (4), 263–282.
Rojas, J.M., Fraser, G., Arcuri, A., 2016. Seeding strategies in search-based unit test generation. Softw. Test. Verif. Reliab. 26 (5), 366–401.
Rojas, J.M., Vivanti, M., Arcuri, A., Fraser, G., 2017. A detailed investigation of the effectiveness of whole test suite generation. Empir. Softw. Eng. 22 (2), 852–893.
Russell, S.J., Norvig, P., 2016. Artificial Intelligence: A Modern Approach. Malaysia; Pearson Education Limited,.
Sahoo, R.K., Ojha, D., Mohapatra, D.P., Patra, M.R., 2016. Automatic generation and optimization of test data using harmony search algorithm. Comput. Sci. Inf. Technol. 23.
Sakti, A., Guéhéneuc, Y.-G., Pesant, G., 2012. Boosting search based testing by using constraint based testing. In: International Symposium on Search Based Software Engineering. Springer, pp. 213–227.
Sharifipour, H., Shakeri, M., Haghighi, H., 2018. Structural test data generation using a memetic ant colony optimization based on evolution strategies. Swarm Evol. Comput. 40, 76–91.
Tracey, N., Clark, J., Mander, K., 1998. Automated program flaw finding using simulated annealing. In: ACM SIGSOFT Software Engineering Notes, vol. 23. ACM, pp. 73–81.
Zheng, Y., Zhang, X., Ganesh, V., 2013. Z3-str: a z3-based string solver for web application analysis. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, pp. 114–124.

Mohammad Nosrati is a PhD student in software engineering at Shahid Beheshti University, Tehran, Iran. He holds a masters degree in software engineering from Shahid Beheshti University. His research interests are software testing, formal methods, machine learning and image processing.

Hassan Haghighi received his Ph.D. in computer engineering from Sharif University of Technology. He is an associate professor at the faculty of Computer Science and Engineering in Shahid Beheshti University, Tehran, Iran. His research focus is on Software Testing, Formal Methods, and Software Architecture.

Mojtaba Vahidi Asl is assistant professor in the Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. He received his BS in computer engineering from AmirKabir University of Technology, MS and PhD degree in software engineering from Iran University of Science and Technology. His research area includes program analysis, software testing and debugging and HCI.