Multi-Objective Evolutionary Algorithm for String SMT Solver Testing

Zhilei Ren*1, Xiaofei Fan*2, Xiaochen Li†3, Zhide Zhou*4
and He Jiang*5
*School of Software, Dalian University of Technology, Dalian, China
†SnT Centre, University of Luxembourg, Luxembourg
{1zren,5jianghe}@dlut.edu.cn, {2Drogon.Fan,4cszide}@gmail.com, 3xiaochen.li@uni.lu,

***Abstract*—String Satisfiability Modulo Theories (SMT) solver is widely used in academia and industry. The runtime efficiency of the solvers may have great impact on various software engineering tasks such as automated reasoning and formal verification. Although many studies have been conducted to test SMT solvers, they mainly focus on detecting soundness bugs of SMT solvers. In contrast, only very few studies concentrate on detecting performance defects of SMT solvers. Moreover, in the existing literatures, we observe two major barriers in generating test cases that trigger performance defects for string SMT solvers, i.e., the guidance information barrier and the diversity barrier. In this paper, we propose a multi-objective evolutionary algorithm, MulStringFuzz, to detect performance defects of string SMT solvers. The unique feature of MulStringFuzz lies in the combination of the multi-objective model and the diversity maintenance mechanism. To tackle the guidance information barrier, MulStringFuzz employs multiple objective functions, i.e., the running time, the code coverage, and the test case complexity to guide the test case generation. To tackle the diversity barrier, a tracing based crowding distance mechanism is proposed to ensure the diversity of generated test cases. Extensive experiments are conducted to evaluate the effectiveness of MulStringFuzz, and we investigate how each proposed mechanism contribute to the overall framework. The test cases generated by MulStringFuzz can cover nearly 5,000 more lines of code and trigger 3.25 times performance defects than StringFuzz, which shows that MulStringFuzz can effectively detect performance defect of the String SMT solver.**

Keywords: String SMT Solvers, Fuzz testing, Multi-objective search algorithm

I. Introduction

String SMT solver is a software tool designed to solve the Satisfiability Modulo Theories (SMT) problems with string constraints, and it has been widely used in many areas, such as software validation [1] and testing [2], [3]. SMT solver errors may lead to crash of the application, or even worse. The performance and correctness of the target application may be influenced by the string SMT solver. For example, in the work of Lucas Bang [4], they use the method of symbol execution to calculate the information leakage of the program. The string solver is used to calculate the path constraints generated by the symbol execution in a run. If the solver has timeout problems or fails to return results, normal analysis results will not be obtained. Hence, it is critical to detect performance defects of the string SMT solver.

Although many approaches have been proposed to test SMT solvers, they mainly focus on detecting soundness bugs of SMT solvers, i.e., when the solver returns the wrong result of instance. For example, STORM [5] detects soundness bugs in the solver by fragmenting the seed test cases, extracting some formulas from them, and eventually constructing satisfiable instances. Dominik Winterer [6] et al. adopt the method called semantic fusion that guarantees satisfiability instances when two existing formulas with known satisfiability are fused into a new formula. Opfuzz [7] also can find a large number of soundness bugs, which proposes type-aware operator mutation. The key idea of type-aware operator mutation is to change the function in the SMT formula with the function that conforms to the type. These studies mainly focus on how to generate instances with known satisfiability to detect SMT solver soundness bugs.

However, only a few work focuses on testing the performance issues of SMT solver. FuzzSMT [8] is the first work which uses grammar-based black-box fuzz testing to complement traditional testing techniques for bit vector and array theory. Alexandra Bugariu [9] proposes a method for string theory, which generates simple formulas with known truth values and derives more complex, equi-satisfiable formulas with their transform rules. Both studies generate a large number of SMT instances, and record test cases that trigger timeouts in the process. In other words, the test cases found are a windfall in testing SMT solvers and they do not propose an effective variant to detect the solver performance defects. To solve this problem, the Z3 developers implement StringFuzz [9], which uses genetic algorithms and takes the solver's running time as the objective function, to generate SMT instances with string constraints. Compared with other approaches, StringFuzz is more effective for finding test cases that trigger timeouts.

Although the existing approaches have yielded very promising results, there still are two major challenges in detecting performance defects of String SMT solvers:

- Guidance information barrier. In the existing studies, the test case generation is modeled as a single-objective problem, i.e., the current methods only use running time as the objective function. Due to the lack of effective guidance information, the search process tends to randomly explore in the solution space. Consequently, such random exploration variant does not guarantee test coverage, and may discard some test cases that potentially trigger solver defects during the search. As a result, the generated test cases cannot be solved within a small running time limit, but their actual running time cannot be predicted.

- Diversity barrier. Shortage of diversity assessment may give rise to the algorithm producing an individual set that all contain the same structural fragments. In genetic algorithms such as StingFuzz, individual genes are exchanged through crossover operations to form new genotypes. In this way, a long-duration structure may be swapped into many individuals, and these individuals take longer to be solved by the SMT solver which leads to the algorithm early convergence.

In order to tackle these challenges, in this paper, we present MulStringFuzz (Multi-objective evolutionary algorithm for String SMT solver testing) to effectively generate instances to detect String SMT solver performance defects. The proposed framework features the combination of the multi-objective and the diversity maintenance mechanisms, to tackle the aforementioned challenges, respectively. On the one hand, to tackle the guidance information barrier, we consider three objectives, i.e., the runtime difference from the reference solver, the code coverage score, and the complexity score, to guide the search process. The basic idea of MulStringFuzz is to use multiple optimization objectives to enhance the guidance information during the search process. We define three optimization objectives as the fitness function: runtime difference of the solver, code coverage score, and complexity of SMT instances. We employ the runtime difference from the reference solver to measure whether a performance defect has occurred. By maximizing the runtime difference between the target solver and the reference solver, we intend to generate test cases that may trigger target solver performance defect. To ensure that more code can be tested in the solution space we explore, code coverage score is another optimization goal. A reward mechanism is designed to encourage instances of overlaying different code snippets during instance generation. In addition, to prevent the test cases from bloating too much, we consider the instance complexity as an optimization objective. Since there are multiple optimization targets, we adopt non-dominated Pareto sort for individual screening. On the other hand, to tackle the diversity barrier, we leverage the similarity between the tracing information collected for test cases to calculate the crowding distance between individuals.

To evaluate our approach, we first compare the effectiveness of MulStringFuzz with the state-of-the-art single-objective approach, i.e., StringFuzz. Second, we verify whether each optimization objective is able to achieve the specified design goal, by evaluating the results obtained by a set of variants of MulStringFuzz. We observe the effect of the optimization target by deleting it during the execution of the algorithm. Finally, we analyze whether using tracing information based similarity could effectively improve the population diversity. Experimental results show that our method can effectively generate test cases for detecting performance defects. There has also been a significant improvement in code coverage and instance size reduction. On average, nearly 5,000 more lines of code can be covered by the test cases generated by MulStringFuzz than StringFuzz; correspondingly MulStringFuzz can trigger 3.25 times performance defects than StringFuzz.

The paper makes the following contributions:

- We propose a fuzzer MulStringFuzz, based on multi-objective evolutionary algorithm, which can automatically generate test cases to detect the String SMT solver's performance defect.
- We conduct extensive experiments to evaluate MulStringFuzz. The results show that the test cases generated by MulStringFuzz can cover more code, and the complexity of test cases is greatly reduced. Using the similarity of tracing information to calculate the crowded distance can effectively reduce the chance of repeatedly instance.

The rest of the paper is organized as follows. Section II provides the background with a motivating example on SMT solver and performance testing. Section III introduces the design of our approach. The evaluation result of our approach is introduced in Section IV. Section V describes related work and Section VI concludes the paper.

II. BACKGROUND AND MOTIVATING EXAMPLE

A. *String SMT solver and Performance Defect*

SMT is a decision problem for first-order theories, which include integers, bit vectors, floating-point, string [10] etc. The SMT solver is a tool to determine whether the formulas in these theories are satisfactory. Z3str3 [11] and CVC4 [12] are two widely used open source solver that implements string theory. The solver verifies the input SMT instance and return SAT/UNSAT. The SMT solver determines that the logic of the instance can be satisfied, and it returns SAT. Otherwise, UNSAT is returned. The performance defect occurs when the SMT solver is unable to get the answer of the instance after a long period of computation.

Developers often refer to performance defect as software bugs [13]. Finding performance defect is also difficult because performance defect have non-failover symptoms. Although the existing work has found a large number of soundness bugs, only few researches focus on performance defect of SMT solver which leads to lack of performance benchmarks. Compared with other well-developed solver theories, string theory has a deficiency in benchmarks to test. In addition, the specialized solver competition SMT-COMP [14] includes a number of solver test benchmarks. However, there are few test cases for string theory, and almost no test cases for detecting performance defect. This still leaves a large gap in the detection of String SMT solver performance defect.

StringFuzz is a genetic algorithm based method, which can generate an instanced that takes a long time for a given solver to solve. This method takes the running time of the target solver as the objective function. In the culling phase of the genetic algorithm, instances are ordered according to a fitness function, selects excellent individuals and removes others. StringFuzz is random in its mutations.

```
(set-logic QF_S)
(declare-fun var3 () Int)
(declare-fun var0 () String)
(declare-fun var2 () String)
(assert (< (str.to.int var2)
        (str.to.int var0)))
(assert (> (str.indexof "+]YAPKJ8b8"
        "8b8" var3) 9))
(check-sat)
```

(a) Individual A.

```
(set-logic QF_S)
(declare-fun var3 () Int)
(declare-fun var0 () String)
(declare-fun var2 () String)
(assert (< (str.to.int var2) 9))
(assert (> (str.indexof "+]YAPKJ8b8"
        "8b8" var3) (str.to.int var0))
(check-sat)
```

(b) Individual B.

```
(set-logic QF_S)
(declare-fun var3 () Int)
(declare-fun var0 () String)
(declare-fun var2 () String)
(assert (< (str.to.int "8b8") 9))
(assert (> (str.indexof "+]YAPKJ8b8"
        var2 var3) (str.to.int var0)))
(check-sat)
```

(c) Individual C.

```
(set-logic QF_S)
(declare-fun var3 () Int)
(declare-fun var0 () String)
(declare-fun var2 () String)
(assert (> (str.indexof "+]YAPKJ8b8"
        var2 var3) (str.to.int var0)))
(check-sat)
```

(d) Individual D.

Fig. 1. Test case generated by MulStringFuzz

B. Motivating Example

In this subsection, we describe the challenges of test case generation for performance defect with an example.

In Figure 1, we present four SMT instances as individuals. Suppose *Individual A* and *Individual B* are generated first through individual crossover or mutation operation as in StringFuzz. *Individual A* takes 0.21077 seconds for the Z3str3 solver to verify. But *Individual B* merely takes 0.20428 seconds. If we only use time as the criterion for single-objective optimization, *Individual B* will be left out of the selection process, and unable to participate in the subsequent evolution process. However, we notice that *Individual B* covers 811 more lines of code at run time than *Individual A*. Interestingly, by swapping node *var2* and node *"8b8"*, we can convert *Individual B* to *Individual C*, which could trigger a timeout of Z3str3. If we consider code coverage information as guidance information, *Individual B* will survive in the population because it performs well in terms of code coverage. In other words, using coverage information as an optimization goal can fully search the solution space and find more structures that trigger defects.

Meanwhile, we continue to investigate *Individual D*, which is transformed from *Individual C* by deleting an assert statement. The structure that triggers the timeout problem is *"(assert (> (str.indexOf "+]YAPKJ8b8" var2 var3) (str.to.int var0)))"* . We note that *Individual D* is simpler, which allowing developers to locate the structure that produces the performance defect more quickly. Hence, we assume that the complexity of the SMT instance could also be utilized as an optimization objective, which would help us reduce the size of test cases. Furthermore, we notice that there may exist conflicts between the complexity of the instance and its corresponding code coverage of the solver. Hence, such phenomena inspire us to adopt multi-object.

Finally, we illustrate the effect of population diversity assessment. Because *Individual D* is better than other individuals, it can constantly participate in the following mutations. If the node is swapped again, *Individual D* generates an entity containing *"(assert (> (str.indexOf "+]YAPKJ8b8" var3 var2) (str.to.int var0)))"*. This individual performs essentially the same as *Individual D* in fitness evaluation (because only two String variable names are exchanged, without structural change), which decreases the diversity of the solutions. Consequently, a diversity assessment is necessary to help find diverse test cases that trigger defects. Moreover, if we introduce the diversity assessment, the similarity between the dynamic execution files of the two individuals is 98.76%. Therefore, we discard one of them to ensure that we generate more timeout use cases with different structures.

III. OUR APPROACH

In this section, we introduce the implementation of MulStringFuzz, include the main framework of MulStringfuzz, the details of individual generation, the fitness function evaluation, the congestion calculation, and the genetic operator.

A. Framework

We implement a multi-objective evolutionary algorithm to find SMT instances that can trigger solver performance defects, shown in Algorithm 1. First we randomly initialize the population with a specified size. In the population, each individual represents a test case. The population is evaluated for fitness and subjected to rapid non-dominated Pareto sequencing [15] in lines 1-5. We set the individual fitness to the Pareto frontier order. The initial population generates the first offspring by 2-tournament selection in lines 6-7.

The algorithm works as follows. In each generation, the offspring are generated from these individuals selected by tournament selection based on dominance. In this process, crossover operators and mutation operators are used to explore

Algorithm 1 MulStringFuzz

```
Input:
    $popSize$: population size;
    $maxGen$: maximum evolutionary generation;
    $cxProb$: crossover probability;
    $mutateProb$: mutation probability;
Output:
    A collection of SMT instances $P_{maxGen}$;
 1: $P_0 \leftarrow$ GeneratePopulation($popSize$)
 2: for each $ind \in P_0$ do
 3:     $ind.fitnesses$ = evaluate($ind$)
 4: end for
 5: $fronts$ = sortNondominat($P_0$)
 6: $Q_0 \leftarrow$ selTournament($fronts, popSize$)
 7: $Q_0 \leftarrow$ GenOffspring($Q_0, muProb, cxProb$)
 8: for each $i \in [1, maxGen]$ do
 9:     $R_i \leftarrow P_{i-1} \cup Q_{i-1}$
10:     for each $ind \in R_i$ do
11:         $ind.fitnesses$ = evaluate($ind$)
12:     end for
13:     $fronts \leftarrow$ sortNondominat($R_i$)
14:     for each $front \in fronts$ do
15:         assignCrowdingDist($fronts$)
16:     end for
17:     $P_i \leftarrow$ selNSGA2($fronts, popSize$)
18:     $Q_i \leftarrow$ selTournamentDCD($P_i$, $popSize$)
19:     $Q_i \leftarrow$ GenOffspring($Q_i, muProb, cxProb$)
20: end for
21: return $P_{maxGen}$
```

the solution space. MulStringFuzz merges the offspring and the parent generation to get the new population in line 9 of algorithm 1. With the new population, the fitness is applied to perform a fast non-dominated Pareto sort [16], in lines 10-13. Then we calculate the crowding distance, select and generate the offspring again, in lines 14-19.

B. Individual definition

In the implementation of genetic algorithms, each individual need to be mapped into the coding space. Each individual corresponds to a unique SMT instance. The individual in this paper is implemented based on SMT-LIB[1] programs. SMT-LIB theory is a standard rigorous description of solver input and output. StringFuzz implements several generation strategies (called generators), which encodes the SMT-LIB instances as abstract syntax trees (ASTs), which consists of four components: *Set logic*, *Declare*, *Assert* and *Check sat*. *Set logic* defines the SMT-LIB theory used by the test cases, which is set to "QF_S" by default. Variables and their types are declared in the second part. For example, lines 2-4 of the example in Fig.1(a) defines two String variables and one Integer variable. Each assert statement is an abstract syntax tree that is stored in the assert list. Line 5 and line 6 in Fig.1(a) respectively represents an assert statement. *Check sat* has a fixed statement which lets the solver check the satisfaction of the test case. Since variables in the assert statement need to be modified in the mutation operator, we use a map to maintain the relationships between variable names and types for type checking.

[1]http://smtlib.cs.uiowa.edu/

C. Design of fitness function

The definition of fitness function plays an important role in the genetic algorithm. This subsection describes the three objective functions, which are the running time difference between the target solver and the reference solver, the code coverage score, and the complexity of test cases.

The runtime difference of the solvers. The first objective function is the runtime difference between the target solver and the reference solver. Runtime is the most immediate manifestation of the performance defects, and the first step when looking for a timeout use case is to define the SMT solver's timeout. Although it is mentioned in the SMT-COMP [14] contest to set 40 minutes as the time-out period for each test case, an excessive time-out period will waste a lot of computing resources when generating test cases. In order to reduce the time elapsed for each test case validation, we choose a relatively small timeout period, so that the overall evolution time is acceptable. Therefore, we do not directly use the running time of the test case as the objective function in the experiment, but using the difference in running time between the target solver and the reference solver as the objective function. If we can maximize the time difference between the two SMT solvers, the SMT instance is more possible to trigger a performance defect in the target solver.

$$tScore(I) = \begin{cases} f_t(I,T), & f_t(I,T) \le f_t(I,B) \\ f_t(I,T) + (f_t(I,T) - f_t(I,B)), & \\ & f_t(I,T) > f_t(I,B) \end{cases} \qquad (1)$$

We use $f_t(I,T)$ and $f_t(I,B)$ to respectively represent the CPU running time of the test case *I* on the target solver T and the reference solver B. Eq. 1 shows how to calculate the running time score of the SMT instance: when the test case *I* uses less time on the target solver than the reference solver, we use the running time of the target solver as the final score; if the running time of the target solver is greater than the reference solver, we combine the time difference between the two running and the running time of the target solver as the final score. For example, the instance in Fig. 1(d) takes 5s to run under Z3str3 (we set 5s as the timeout period in our experiment) and 0.01731 second to run under CVC4. So the $tScore$ of this instance is $5 + (5 - 0.01731) = 9.98269$. Using this incentive mechanism can expand the running time difference between the two solvers, we can obtain test cases that are difficult to be solved by the target solver.

Code coverage score. The intention of this objective is to examine as much code path as possible to trigger performance defects. However, we need to pay attention that the SMT solver is a highly complex system. For instance, the z3 solver has

Algorithm 2 CalCodeCoverageScore

```
Input:
    $gcdaDir$: code coverage information directory;
    $codeInfos$: global code coverage information file;
Output:
    Code coverage score $covSocre$;
1: $covInfos \leftarrow$ collectGcovInfo($gcdaDir$)
2: $covSocre \leftarrow 0$
3: for each $fileInfos \in covInfos$ do
4:     if $fileInfos \notin codeInfos$ then
5:         $covSocre$ += 1000
6:     else
7:         for each $row \in fileInfos$ do
8:             if $row \notin codeInfos[fileInfos]$ then
9:                 $covSocre$ += 100 $\times fileInfos[row]$
10:            end if
11:            if $row \in codeInfos[fileInfos]$
               and $fileInfos[row] > codeInfos[row]$ then
12:                $covSocre$ += 10 $\times fileInfos[row]$
13:            else
14:                $covSocre$ += 1
15:            end if
16:        end for
17:    end if
18: end for
19: return $covSocre$
```

216,721 lines of code (version 4.8.10, obtained by gcov test), and the number of lines covered by a test file only which is a small fraction of the entire code lines (i.e., the code coverage rate is very low). As a result, a change on the test case (assert statement change or node change) may only affect the coverage of a few lines of code in the solver, which has no significant impact on the calculation of code coverage. Hence, the common code coverage testing tools [17]–[20] may not be suitable to measure the slight changes of the code coverage. To solve this problem, we designed a reward mechanism to encourage new code statements or test cases with more runs after mutations.

As shown in algorithm 2, we use a reward mechanism to calculate the code coverage score. We can get "*.gcda" files with coverage information after program Instrumentation. We iterate over these files to get statement coverage information, which includes names of the solver's code files executed by the test case and the execution times of each statement in the code file. Each time we run MulStringFuzz, we maintain a static dictionary $codeInfos$ to store the maximum number of execution times of each statement of all code files. For each test case, we obtain the execution times of each statement, which is saved in $covInfos$. Then we compare $covInfos$ with the static dictionary $codeInfos$ and calculate bonus scores. We iterat through the names of code files that appear in $covInfos$, if a new code file name appears, it means that this test case greatly expands the covered code statement. In order to encourage this situation, we give it a bonus of 1,000 scores. Then we iterate over each statement in the file: 1) If a new code statement appears, we assign a bonus socre as the execution times $\times$ 100 to this code statement. 2) If the code statement has already appeared in $codeInfos$, but the execution times of the test case is bigger than the execution times of $codeInfos$, the bonus points are the augmentation of the execution times $\times$ 10; 3) In other cases, each statement has a bonus score of 1. After we calculate the bonus score for each code statement, the newly-appearing code file name and the maximum execution times are updated into $codeInfos$. The sum of all the bonus scores is the code coverage score for this test case.

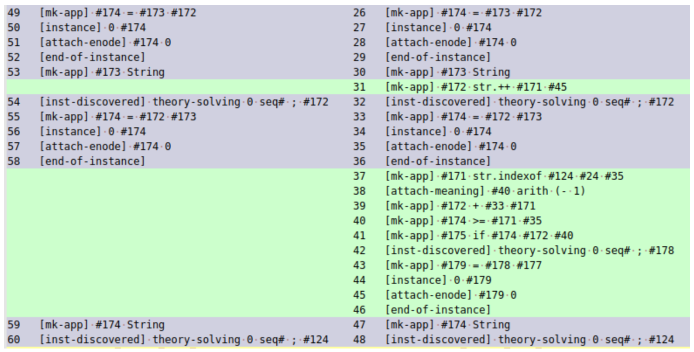
Complexity of test cases. Finally, in order to prevent excessive bloating of test cases, our third objective function is to constrain the complexity of test cases. This goal aims to improve the readability of test cases and make it as easy as possible for developers to find bugs. We find that the main reason for complex test cases are the number of assertion statements and the depth of the nesting. Therefore, we cannot simply use the size of the test case or the number of assert statements to measure the complexity of the test case. In order to comprehensively consider the number of assert statements, the depth of nesting, and the number of nodes, we use Eq. 2 to calculate the complexity of an SMT instance.

$$comScore(I) = 0 - \sum_{i=1}^{len_{assert}} \sum_{j=1}^{deep_{assert_i}} j * num_{node} \tag{2}$$

Eq. 2 uses the number of nodes in assertions to calculate the complexity of the test case. In this way, the depth of each node in the AST tree can be weighted into the result. It helps reflect the actual complexity. For example, if the node is at the level *n* of the tree, it is considered that the node accumulates *n* points. Then we sum the points of each node to get the final score. For example, the individual complexity in Fig. 1(c) is $(1 \times 3 + 2 \times 1) + (1 \times 3 + 2 \times 3 + 2 \times 1) = 16$, and the individual complexity in Fig. 1(d) is 11. Finally, we use the negative of the score as a complexity score for convenience to find the maximum value.

D. Diversity assessment

The diversity assessment is to enable the algorithm to generate more diverse individuals during the search process. In the classical NSGA2 algorithm, crowding distance and crowdedness comparison operator are often used as the criteria for the comparison between individuals in the population, so that the individuals in the quasi-Pareto domain can evenly extend to the whole Pareto domain and ensure the diversity of the population. The crowding distance calculation of NSGA2 algorithm is: sort the population in ascending order according to the size of each objective function value; for each objective function, the boundary solution (the solution with maximum and minimum values) is specified as the value of an infinite distance; All other intermediate solutions are specified as equal to the absolute difference of the normalized values of the two



```
49  [mk-app]·#174·=·#173·#172                          26  [mk-app]·#174·=·#173·#172
50  [instance]·0·#174                                  27  [instance]·0·#174
51  [attach-enode]·#174·0                              28  [attach-enode]·#174·0
52  [end-of-instance]                                  29  [end-of-instance]
53  [mk-app]·#173·String                               30  [mk-app]·#173·String
                                                       31  [mk-app]·#172·str.++·#171·#45
54  [inst-discovered]·theory-solving·0·seq#·;·#172     32  [inst-discovered]·theory-solving·0·seq#·;·#172
55  [mk-app]·#174·=·#172·#173                          33  [mk-app]·#174·=·#172·#173
56  [instance]·0·#174                                  34  [instance]·0·#174
57  [attach-enode]·#174·0                              35  [attach-enode]·#174·0
58  [end-of-instance]                                  36  [end-of-instance]
                                                       37  [mk-app]·#171·str.indexof·#124·#24·#35
                                                       38  [attach-meaning]·#40·arith·(-·1)
                                                       39  [mk-app]·#172·+·#33·#171
                                                       40  [mk-app]·#174·>=·#171·#35
                                                       41  [mk-app]·#175·if·#174·#172·#40
                                                       42  [inst-discovered]·theory-solving·0·seq#·;·#178
                                                       43  [mk-app]·#179·=·#178·#177
                                                       44  [instance]·0·#179
                                                       45  [attach-enode]·#179·0
                                                       46  [end-of-instance]
59  [mk-app]·#174·String                               47  [mk-app]·#174·String
60  [inst-discovered]·theory-solving·0·seq#·;·#124     48  [inst-discovered]·theory-solving·0·seq#·;·#124
```

Fig. 2. Trace log comparison between *Individual C* and *Individual D*.

adjacent solutions; the same calculation method is adopted for other objective functions. All the crowding coefficient values are calculated by adding the distance values of each target of the individual, and each objective function will be normalized before calculating the crowding coefficient. However, in our algorithm, we do not expect individuals to be uniformly distributed in the whole Pareto domain. We introduce two other objective function to increase the search scope and reduce the complexity of the test cases, and we would prefer to have more test cases with timeouts. In order to ensure the diversity of the population, we do not use the traditional method to calculate the crowding degree, but used the similarity of the dynamic execution files of each individual to represent the crowding distance.

$$sim(I_1, I_2) = \frac{2 * \|I_1 \bigcap I_2\|}{\|I_1\| + \|I_2\|} \tag{3}$$

$$crowdingdist(I) = 1 - \frac{\sum_{O}^{O \in (fronts - I)} sim(I, O)}{\|fronts - I\|} \tag{4}$$

Trace log comparison is used to measure the similarity between newly generated test files and seed files, which is proposed by Opfuzz's work [7]. As shown in Fig. 2, the trace log records runtime information and provide a detailed picture of the target program's runtime behavior [21], [22]. So it can be used to represent the similarity between instances. We use this concept to measure the similarity between the two test case, the specific calculation method between Eq. 3. $\|I\|$ represents the number of lines of the trace log, and $I_1 \bigcap I_2$ represents the same portion of two trace logs which is computed by diffscope[2]. We obtained the Pareto front surface through Pareto sorting, and then used Eq. 3 to calculate the similarity between individuals. For example, we calculate the similarity between *individual C* and *individual D* in Fig. 1. The tracing log of *individual C* contains 4,553,000 lines, and the tracing log of *individual D* contains 4,430,457 lines. Through diffoscope comparison, the same lines of two tracing log is 1,680,940. Then the similarity of these two individuals is: $(2 \times 1,680,940) \div (4,553,000 + 4,430,457) = 0.374$. In order to synchronize with the concept of crowding distance in NSGA2, an individual with a larger crowding distance indicates a smaller density of surrounding solutions. We use one minus the average similarity as the crowding distance of an individual, as Eq. 4 shows.

[2]https://diffoscope.org/

E. Genetic operator

The exploration of solution space with genetic algorithm mainly uses three kinds of operators: the selection operator, the mutation operator and the crossover operator. We explain how to use these three operators in our research.

Selection operator. Selection is a necessary operation in population evolution. The functionality of selection is to eliminate individuals with poor fitness. Only those individuals who perform well have the chance to survive. MulStringFuzz uses two selection operators:

- Tournament selection. The crowding is not calculated for the first generation population, so the order of the Pareto front is used as fitness. Take two individuals from the population at a time, and select the ones which's fitness well.
- Tournament selection based on dominance between two individuals. After a quick calculation of non-dominant ordering and crowding, each individual I in the population has two properties: the non-dominant ordering determined by the non-dominant ordering and the crowding. In either case, individual I will prevail: the individual I is located in the non-dominant layer superior to that of individual J, or the two individuals do not inter-dominate but the crowding distance of individual I is greater than individual J's crowding distance.

Crossover operator. Crossover operation is to imitate the mating process in nature. Two individuals in a population can exchange each other's genes through sexual reproduction, thereby generating new genotypes. This process can increase the diversity of population genotypes. In order to ensure a sufficient amount of crossover, we select 20% of the sum of the number of assert statements in the two test cases for exchange. If the number of asserts of a parent does not reach the number of exchanges, then two individuals swap one assert statement.

Mutation operator. If only the crossover operation is performed, it may lead to premature convergence. Because the crossover operator can only change the genotype of the individual, and does not generate new genes. Based on the work of StringFuzz, we have defined the basic mutation operators:

- New variable: We add a new type of variable, and generate a statement using the variable.
- Delete variable: We delete a variable and replace the place where the variable is used in the assert statement with other variables. In order to ensure the availability of the variable deletion, it is necessary to ensure that there is at least one variable of each type when the variable is deleted.
- Add assertion: We add a random assertion. When adding a assertion, we need to check the validity of the variables.

- Delete assertion: We delete a assertion randomly;
- Fuzz numbers and characters: We randomly change the string or number that appears in the test case.
- Replace Type: We replace the modifiable function in Assertion with a function of the same type without modifying the contents.
- Exchange the same node: We iterate through the assertion list, and randomly pick two identical nodes to swap their positions.

IV. EVALUATION

A. Research Questions

This study focuses on the following research questions (RQs):

- RQ1: Is our tool more effective at generating test cases to trigger performance defects?
- RQ2: Do these three objective functions help us discover the solver's performance defects?
- RQ3: Can population diversity be improved by using file similarity to calculate crowding distance?
- RQ4: How to select the population size, crossover probability, and mutation probability in the algorithm?

Among these RQs, RQ1 evaluates the effectiveness of our tool. We illustrate the effectiveness of our tool by comparing it with StringFuzz for single-objective optimization, and by the effect of the generated test cases. the purpose of RQ2 is to verify whether each objective function we choose achieves the desired effect. We analyze the effect of each objective function by comparing the result sets generated by the different variations. RQ3 is to discuss population diversity. To increase the diversity of the generated test case set, we use the test file trace log similarity to calculate the crowding distance. We compare the Pareto surface formed by different variations generate. The purpose of RQ4 is to discuss the influence of parameter selection on the experiment. To illustrate the effect of population size, we record the convergence time of the algorithm under different population sizes and the number of test cases that trigger solver timeout in the generated result set. We find the recommended population size by comparing the different trends in the two recorded values.

B. Experimental Setup

Our experiments run under a PC with an Intel Core i9 2.8 GHz CPU, 32 GB memory, and Ubuntu 20.04.2 LTS. We set the size of the population as 40, the crossover probability as 0.75, and the mutation probability as 0.25 when running the multi-objective genetic algorithm. The experiment select two of the most popular open source solvers. We use Z3str3 as the target solver (version 4.8.10), CVC4 is selected as the reference solver (version 1.6). We do not choose Z3str4 because it is a meta-solver, which uses algorithm selection techniques to complete the instance solver for unforeseen problems. Each solver uses the default settings during the experiment, and memory to run each solver is limited to 8GB. All programs run in single-threaded mode. The output format of the test case is SMT-LIB2.5.

In the experiment, we compare MulStringFuzz and StringFuzz. At the same time, we design several different variants to verify the effectiveness of the method. Two methods are used in the experiment, which are StringFuzz based on single-objective optimization and MulStringFuzz based on multi-objective optimization.

For StringFuzz we design two variants: Default and Runtime-Difference.

- Default: The Default variant uses the elapsed time of a single solver as the optimization goal and uses random mutation variant during iteration.
- Runtime-Difference: The Runtime Difference variant takes the time difference between two solvers as the optimization target, and the default setting of StringFuzz is used for the rest of the departments.

There are five variations for MulStringFuzz: Default, Only-Time, No-Coverage, No-Complexity, Default-Crowding.

- Default: The Default variant uses three objective function: the running time difference between the target solver and the reference solver, the code coverage score, and the size of the test case. The individual crowding distance is calculated using the similarity of the dynamic trace files of the test cases.
- Only-Time: The only-time variant uses the target solver run time as the optimization goal. Other policies are the same as the Default policy.
- No-Coverage: The No-Coverage variant does not use code coverage as an optimization goal.
- No-Complexity: The No-Complexity variant does not use the complexity of test cases as an optimization goal.
- Default-crowding: The crowding distance between individuals is calculated using the default variant in NSGA2.

Since genetic algorithm has a high degree of randomness, in order to make the experimental results more accurate, we ran each variant for 24 hours to generate a result set and repeated each variant 50 times.

C. RQ1: Performance of MulStringFuzz

To answer this RQ, we compare the test case result sets generated by MulStringFuzz and StringFuzz. We illustrate the effectiveness of each optimization goal by comparing the result sets generated by different variations. Tot_{lines} represents the lines of code covered by all generated test cases in an algorithm run, measured by a static file maintained at each run. After each file is executed, we count the code it covers and updates it to the static file. At the end of each run, we can use this file to measure how many lines of code are covered in one run of the algorithm. Avg_{lines} represents the average number of lines of code covered by each test case in the result set. Avg_{tScore} represents the average time score. Note that StringFuzz's Default variation and MulStringFuzz's Only-Time variation take the time of a single solver as the optimization goal, so its average time score is smaller than 5s. Other variations use the time difference proposed by us, so the score will be higher than 5s. We also measure the

TABLE I
RESULTS UNDER DIFFERENT VARIATIONS

Method	Variant	Tot_{lines}	Avg_{lines}	Avg_{tScore}	$Num_{timeout}$	$Avg_{complexity}$
StringFuzz	Default	15,332.1	13,011.8	4.33	4	44.07
	Runtime-difference	18,268.5	13,048.7	4.78	5	48.87
MulStringFuzz	Default	21,188.5	13,593.5	7.85	13	22.88
	Only-Time	20,093.2	13,496.6	4.25	6	21.26
	No-Coverage	18,326.2	13,127.5	5.93	5	20.5
	No-Complexity	22,161.1	16,434.2	8.98	12	95.5
	Default-Crowding	21,570.6	13246.0	7.56	10	26.03

running time of the test cases in the result using the SMT-COMP standard (2,400s). $Num_{timeout}$ is the number of test cases that trigger a 2,400s timeout in each result set. The final metric is $Avg_{complexity}$, which is the average complexity of the result set.

As we can see from Table I, the test cases generated by MulStringFuzz cover 5000 more lines of code than those generated by stringFuzz. MulStringFuzz can generate 9 more test cases with timeouts than StringFuzz at the same run time. This shows that using code coverage information helps explore more structures and generate more test cases which trigger timeout of solver. At the same time, the average complexity of the result set generated by our method is only 22.88, which is much lower than StringFuzz's 44.07. This means the test cases generated by MulStringFuzz can reduce two assert statements. The test file that MulStringFuzz generated retains the structure of triggering solver timeouts while removing some statements that are not related to triggering performance defects.

We use cumulative distribution function (CDF) diagrams to visualize the results. 2,400s is used as the timeout time to measure the test cases in the result set. We sort all times from small to large, draw points (t_1, 1), (t_2, 2) etc., and in general (t_k, k). By normalizing the Y-axis to [0, 1] (without dismissing the timeout case), we can approximate the cumulative distribution function. CDF diagrams are used in Satzilla's work [23] to describe solver performance. The point in the CDF figure can be interpreted as "what is the probability of solving a random problem in *t* seconds". To make the trend of the image more apparent, we use logarithmic coordinates to represent the time.

As shown in Fig. 3, the result set generated by MulStringFuzz contains some instances that are relatively short in time. 70% of the test cases could be solved in 1s. Because our algorithm contains multiple objective functions, these individuals may be superior in terms of code coverage score and individual complexity, and therefore have poor performance in terms of time. The StringFuzz generates more test cases distributed between 5 seconds and 600 seconds. Due to the StringFuzz algorithm takes the solver running time as the

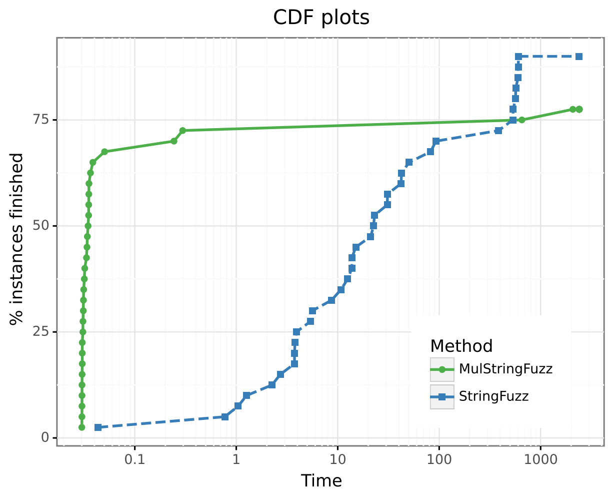


Fig. 3. CDF plots of the result set.

objective function, the individuals retained are those whose running time exceeds 5s. With the running time limit of 2,400s, Z3str3 solved only 77.5% of the result set of MulStringFuzz, less than 90% of the result set of StringFuzz. MulStringFuzz could find more instances that trigger the timeout of the solver. We report the test cases to the developers of Z3str3 to help them check and resolve the performance defects.

The experimental results show that MulStringFuzz can trigger 3.25 times performance defects than StringFuzz. The test cases generated by MulStringFuzz can cover more code statements and have a lower complexity. We believe that MulStringFuzz is better than StringFuzz at detecting solver performance defects.

D. *RQ2: Validity of optimization objectives*

In this subsection, we verify whether each objective function we choose achieves the desired effect. First, we analyze whether the difference in running time with two solvers helps find more test cases that trigger timeouts. We can see the variant that uses the time difference as objective function

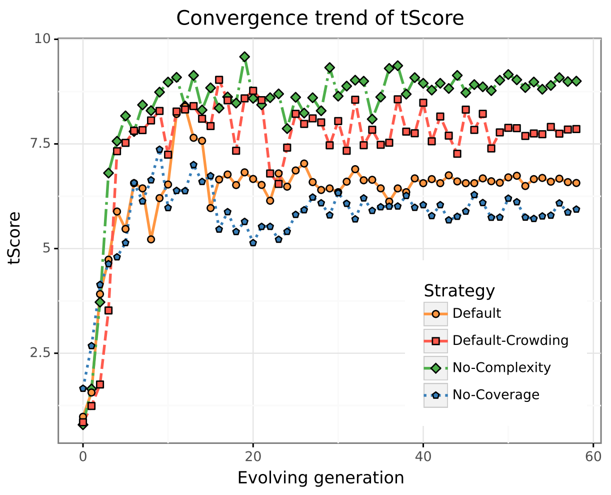


Fig. 4. Convergence tend of tScore

generates more timeout test cases than the variant that uses a single solver run time as objective function. When we use the single solver time as objective function, the average time score of the population is very close to full score (because the timeout is 5s). However, a few test cases trigger timeout under the running time limit of 2,400s. This shows that the variant of using a single solver run time can indeed produce individuals who perform well within the specified timeout range, even if we can set the timeout large enough, we can also find test cases that can trigger performance defects of String solver. This is a big test for the efficiency of the program, which needs to consume a lot of computing resources. Using the time difference policy can increase the probability of finding test cases that trigger performance defects.

Second, we observe whether using code coverage score can effectively expand the search solution space. We can see that the result sets generated by the Default and No-Complexity variant triggered timeouts with 13 and 12 cases respectively, which is far more than the 5 cases generated by the No-Coverage variant. The variant of using code coverage to guide the search explored more lines of code during the algorithm's run and gain the advantage in the number of timeout test cases. The No-Coverage variant equivalent to the algorithm of random selection and random evolution. It explores the solution space aimlessly, relying on the computing power of the computer to find test cases that may trigger performance defects. Some test cases can be solved in a relatively short time but cover more lines of code statement. MulStringFuzz allows these test cases to survive in the population, which allows the algorithm to explore more structures. It helps MulStringFuzz to generate more test cases that trigger performance defects.

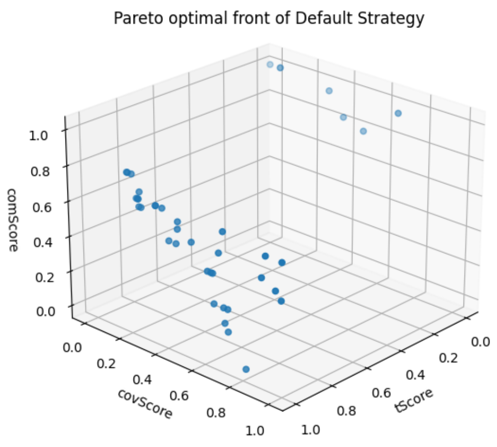
Finally, we compare the effect of using complexity as the objective function. The average file complexity under the No-Complexity variant is 95.5, which is much higher than the other variations. However, under this variant, the total number of lines of code covered during the run and the average number of lines of code covered per file in the result set is the highest. There is a conflict between complexity and code coverage. A test case with a complex structure can cover more code statements. Therefore, without the complexity constraint, the algorithm can search for more solution space. Overly complex test cases cause difficulties for developers identifies defects. Although using complexity as the objective function reduce the number of timeout test cases, it can significantly reduce the complexity of test cases.

We verify the effect of three objective functions through compare the result generated by different variants. The time difference between the target solver and the reference solver helps find SMT instances that are more difficult for the target solver. Using code coverage information to guide the search process allow MulStringFuzz to find more test cases that trigger performance defects. We can clearly see that using complexity as objective function can prevent test case bloat. The three objective functions we selected are effective.

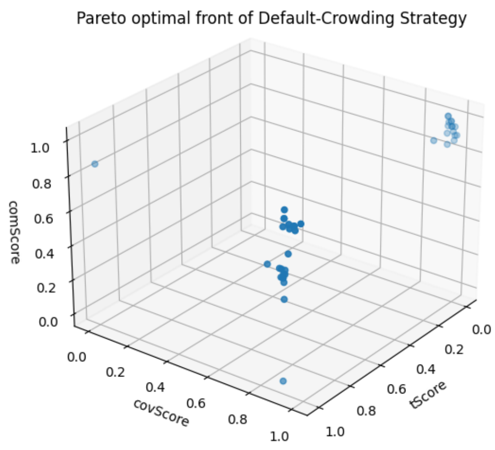
E. *RQ3: Effect of individual crowding distance*

In order to explain the influence of the two different crowding distances on the algorithm, the convergence speed and the distribution of the final result set are used to illustrate. Fig. 4 shows the variation trend of the average $tScore$ of the population as the population generation changes during the operation of the algorithm. It can be seen that when the custom crowding distance is used, the average time score of the crowd is relatively high, which can reach about 7-9, and finally converges to about 9. When the default crowding distance is used, the average time score is around 7-8.5 and eventually converges to about 8. However, the convergence rate of the two is relatively close, and the change of $tScore$ tends to be flat from the 45th generation to the 55th generation. The default policy does not change the convergence rate of the algorithm, which only affects the resulting set. Using the default policy can help increase the average running time score of the population. As you can also see from Table 1, the result set generate under the default policy contains more test cases that trigger long timeouts. The same effect as the no-Complexity variant can be achieved, and the complexity of the test case is superior to the no-Complexity variant. So we think the default policy is better at generating test cases that trigger performance defects of solver.

In addition, we analyze the distribution of the result set. Fig. 5 shows the Pareto frontage generated by both the Default variant and the Default-crowding variant. Each individual in the result set is evaluated for fitness and each goal is normalized. (Although we use a negative comScore score, we normalize it to $[0, 1]$). Since there are three objective functions, the figure is shown as a three-dimensional graph. Fig. 5(a) is the Pareto preface generated by using the Default-Crowding variant. We can see that the test cases are concentrated in the space . The solution dispersion obtained by the Default variant is relatively scattered, and it can be seen that we have more individuals with the score of solution concentration time.

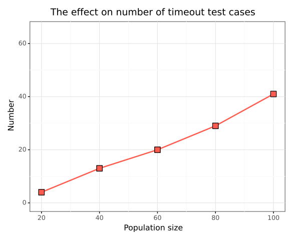


(a) Default variant.

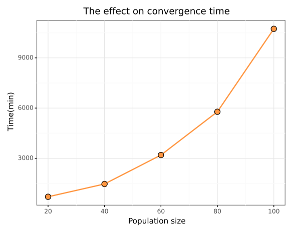


(b) Default-Crowding variant.

Fig. 5. Pareto frontiers generated by two different variations



(a) Timeout test cases number.



(b) Convergence time.

Fig. 6. Convergence time and timeout test cases number in different population sizes.

We believe that using the default crowding distance calculation does not fail to represent the actual population diversity. Trace log can effectively reflect the execution trajectory of a test case, so it is more accurate to judge the similarity of two test cases. However, it is important to note that calculating the similarity between test files is computationally expensive. When running both variations with the same parameters, the Default one takes twice as long as default-crowding. Due to the trace log are large, it take a lot of time to calculate the same number of lines for two trace logs.

F. RQ4: Parameter sensitivity analysis

Preliminary experiments show that compared with mutation probability and crossover probability, population size has a greater influence on parameters, so the influence of population size is analyzed in this RQ.

The size of a population is the number of individuals per generation. If there are more individuals in the population, that means there is a greater chance of finding the optimal solution. However, the population contains more individuals, the more computational resources are used to calculate fitness and population evolution. To illustrate the effect of population size, we record the convergence time of the algorithm under different population sizes and the number of test cases that trigger solver timeout in the generated result set. In order to facilitate comparison, we calculate the relative measure of the results obtained when the population size is 40, As shown in Fig. 6. When the population size is less than 60, the convergence time grows linearly as the number of timeout test cases increases. When the population size is greater than 70, although the number of timeout test cases still increase, the increase in convergence time is more pronounced. So we spend more computing resources, but the efficiency of the algorithm is not significantly improved.

By comparing the trend of convergence time and the number of timeout test cases, we believe that the appropriate range of population size is $[40, 60]$. The population size within this interval can maximize the use of computing resources.

V. THREATS TO VALIDITY

After analyzing the experimental results, there are three possible threats to the effectiveness of our experiment. First, the goal of this paper is to detect the performance defects of the solver, so we need to consider the impact of computing resources on the results of running experiments. The stronger CPU and larger memory will make the solver solve test cases faster. But in the actual development, these computing resources are also limited. If the benchmark solver can get the result of the test case running but the target solver cannot solve under the same computing resources, the test case will trigger the performance defect of the target solver. Therefore, in our experiment, we choose the same hardware environment and limit the running memory of each solver to 8GB. Moreover, in the process of experimental evaluation, we do not consider whether the choice of the target solver will affect our experimental results. Our optimization goal includes code coverage score so we need to use gcov to compile the target solver which will reduce the number of solvers that we can choose. In order to minimize this threat, we choose z3str3 as the target solver of our experiment which is widely used in open source solvers. In addition, the test cases generated in our experiment are based on SMT-LIB 2.5 syntax, which is one of the versions of SMT-LIB. Due to the syntax differences between various versions of SMT-LIB, the syntax structure of test cases may be different. We choose SMT-LIB2.5 which is compatible with most of the current solvers in our experiment. For other versions of SMT-LIB, we only need to change the syntax generation part, which does not affect other parts of the algorithm or the effectiveness of our method.

VI. Related Work

A. *SMT solver testing*

SMT solver plays a vital role in many fields such as formal verification [24], program analysis [25], and software testing [26]. So far, a lot of work pay attention to test the SMT solver. FuzzSMT [8] focuses on finding SMT solver crashes of bit-vector and array instances, which uses syntax-based black-box fuzzing to trigger crash instances. Some work applies fuzzy testing based on search [27], [28] , coverage guidance [29], [30] and symbolic execution [25] to test solvers. StringFuzz looks for performance defects in string logic by changing the transformation benchmark case or by randomly generating formulas from the syntax. Coverage-guided fuzziness testing is applied to floating point constraints in the Just Fuzz-It Solver (JFS) tool [31]. In terms of floating point constraint resolution, JFS can either compete with or complement advanced SMT solvers, and JFS's cover-guide approach provides significant benefits over naive fuzzing in floating point domains.

There is also a lot of work focusing on finding solver soundness errors. They test the solver mainly by constructing SMT instances with known satisfiability. If the solver returns an incorrect result, a robustness error is triggered. In Storm's research [5], some formulae satisfying the known row are obtained by dividing the seed test cases, and then they are rewritten and constructed into test cases. In addition, test cases are generated by semantic fusion and type-aware operation mutation, which achieves good results. Some work focus on how to test the solver's various API parameters and options for running the solver. Cyrille Artho [32] uses model-based testing (MBT) to test the sequence of API calls and different system configurations. Niemetz [33] develope a model-based API fuzzing framework to detect API usage problems of SMT solvers. Different from these work, MulStringFuzz combines fuzzing test with multi-objective evolutionary algorithm to improve the efficiency of detecting the performance defects of String SMT solvers.

B. *Multi-objective evolutionary algorithm*

The main task of a multi-objective evolutionary algorithm is to solve multi-objective optimization problems by evolutionary computation. The evolutionary algorithm can explore the solution space randomly, so it fits the software testing field very well. In this process, multiple optimization objectives can be designed to achieve better testing results. Multiobjective genetic algorithms based on Pareto sorting have another key point: the algorithm is looking for a set of Pareto solutions (not a single Pareto solution), so the diversity of the population is better. Sapienz [34] introduces multi-objective optimization into Android application testing, which uses minimized length to optimize test sequences while maximizing test coverage and fault detection. Stoat's work [35] also focuses on Android testing. Its goal is to maximize code-level coverage while increasing the variety of tests. Wuji [36] also applied the multi-objective evolutionary algorithm to the field of game testing. They a combination of evolutionary algorithms and DLR to test more game states and cover more lines of code. In our work, MulStringFuzz uses a multi-objective evolutionary algorithm for increase guidance information during search, towards covering more code statements and finding more test cases that trigger performance defects.

VII. Conclusion

In this paper, we propose a multi-objective evolutionary algorithm, MulStringFuzz, which is used to generate test cases to detect solver performance defects. We use three optimization objectives to increase guidance information during the algorithm search. The run time difference between the solver and the reference solver helps us find test cases that trigger solver performance defects. We use code coverage scores to encourage the algorithm to explore more solution space and introduce complexity constraints on SMT instances to prevent test case bloat. We set up different variations to verify the effectiveness of each optimization goal. The experimental results show that our algorithm can generate more test cases that trigger the solver timeout, which can effectively detect the solver's performance defects. On average, nearly 5,000 more lines of code can be covered by the test cases generated by MulStringFuzz than StringFuzz, correspondingly MulStringFuzz can trigger 3.25 times performance defects than StringFuzz. Importantly, the complexity of the individual in the generated result set can be effectively reduced. At the same time, we use the similarity of trace log to measure the crowding distance between individuals, which can expand the diversity of individuals in the population to a certain extent.

For future work, the method in this paper can be generalized to the theory of other solvers to find performance defects. We also plan to modify the calculation method of the time score to find the test cases that allow multiple solvers to time out simultaneously, and the generated test cases can be used to extend the test benchmark of SMT-COMP. At the same time, because the measurement of code coverage information is time-consuming, we plan to speed up the calculation through multi-threading, to improve the search efficiency.

Acknowledgment

This work is supported in part by the National Natural Science Foundation of China under grant no. 61772107, 62072068, and 62032004.

References

[1] M. Madsen and E. Andreasen, "String analysis for dynamic field access," in *Compiler Construction*, A. Cohen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 197–217.

[2] H. Samirni, M. Schafer, S. Artzi, T. Millstein, F. Tip, and L. Hendren, "Automated repair of html generation errors in php applications using string constraint solving," *Proceedings - International Conference on Software Engineering*, pp. 277–287, 06 2012.

[3] C. Park, H. Im, and S. Ryu, "Precise and scalable static analysis of jquery using a regular expression domain," *ACM SIGPLAN Notices*, vol. 52, pp. 25–36, 11 2016.

[4] L. Bang, A. Aydin, Q. Phan, C. S. Pasareanu, and T. Bultan, "String analysis for side channels with segmented oracles," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds. ACM, 2016, pp. 193–204.

[5] M. Mansur, M. Christakis, V. Wüstholz, and F. Zhang, "Detecting critical bugs in smt solvers using blackbox mutational fuzzing," 11 2020, pp. 701–712.

[6] D. Winterer, C. Zhang, and Z. Su, "Validating smt solvers via semantic fusion," 06 2020, pp. 718–730.

[7] D. Winterer, C. Zhang, and Z. Su, "On the unusual effectiveness of type-aware operator mutations for testing smt solvers," *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 1–25, 11 2020.

[8] R. Brummayer and A. Biere, "Fuzzing and delta-debugging smt solvers," *ACM International Conference Proceeding Series*, pp. 1–5, 01 2009.

[9] D. Blotsky, F. Mora, M. Berzish, Y. Zheng, I. Kabir, and V. Ganesh, "Stringfuzz: A fuzzer for string solvers," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 45–51.

[10] A. Biere, M. Heule, H. Maaren, and T. Walsh, "Handbook of satisfiability: Volume 185 frontiers in artificial intelligence and applications," 01 2009.

[11] L. de Moura and N. Bjørner, "Z3: an efficient smt solver," vol. 4963, 04 2008, pp. 337–340.

[12] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "Cvc4." 01 2011, pp. 171–177.

[13] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," ser. PLDI '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 77–88.

[14] "Smt-comp 2021," [EB/OL], https://smt-comp.github.io/2021/ Accessed May 31, 2021.

[15] K. Deb and K. Deb, *Multi-objective Optimization*. Boston, MA: Springer US, 2014, pp. 403–449.

[16] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii," in *Parallel Problem Solving from Nature PPSN VI*, M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. J. Merelo, and H.-P. Schwefel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 849–858.

[17] M. Bohme, T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, 12 2017.

[18] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," 10 2018, pp. 2095–2108.

[19] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "Cerebro: context-aware adaptive fuzzing for effective vulnerability detection," 08 2019, pp. 533–544.

[20] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 579–594.

[21] R. Khoury, S. Hallé, and O. Waldmann, "Execution trace analysis using ltl-fo," in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, T. Margaria and B. Steffen, Eds. Cham: Springer International Publishing, 2016, pp. 356–362.

[22] M. Fischer, J. Oberleitner, H. Gall, and T. Gschwind, "System evolution tracking through execution trace analysis," in *13th International Workshop on Program Comprehension (IWPC'05)*, 2005, pp. 237–246.

[23] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: portfolio-based algorithm selection for sat," *Journal of artificial intelligence research*, vol. 32, pp. 565–606, 2008.

[24] R. DeLine and K. R. M. Leino, "Boogiepl: A typed procedural language for checking object-oriented programs," 2005.

[25] S. Gulwani, S. Srivastava, and R. Venkatesan, "Program analysis as constraint solving," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 281–292.

[26] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, Dec. 2008.

[27] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin, "Deploying search based software engineering with sapienz at facebook," in *Search-Based Software Engineering*, T. E. Colanzi and P. McMinn, Eds. Cham: Springer International Publishing, 2018, pp. 3–45.

[28] G. Fraser and A. Arcuri, "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*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 416–419.

[29] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–177.

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

[31] D. Liew, C. Cadar, A. F. Donaldson, and J. R. Stinnett, "Just fuzz it: Solving floating-point constraints using coverage-guided fuzzing," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 521–532.

[32] C. Artho, A. Biere, and M. Seidl, "Model-based testing for verification back-ends," in *Tests and Proofs*, M. Veanes and L. Viganò, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 39–55.

[33] A. Niemetz, M. Preiner, and A. Biere, "Model-based api testing for smt solvers," in *Proceedings of the 15th International Workshop on Satisfiability Modulo Theories, SMT*, 2017, pp. 24–28.

[34] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 94—-105. [Online]. Available: https://doi.org/10.1145/2931037.2931054

[35] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 245–256.

[36] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 772–784.