2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)

Specification-based Test Case Generation with Constrained Genetic Programming

Yuji Sato
Faculty of Computer and Information Sciences
Hosei University
Tokyo, Japan
yuji@k.hosei.ac.jp

***Abstract*—Since current specification-based testing (SBT) faces some challenges in regression test case generation, we have already proposed a new method for test case generation that combines formal specification and genetic algorithms (GA). This method mainly reconfigures formal specifications though GA to generate inputs data that can kill as many as possible mutants of the target program under test. In this paper, we propose ideas to improve the operability and the accuracy of solution search of this method. Specifically, we propose a specification-level constrained operation using genetic programming and discuss effectiveness from the viewpoint of clarity of chromosome notation and ability to search for solutions.**

Keywords*—test data generation, genetic programming, specification-based testing, regression testing, mutant testing

I. Introduction

Functional scenario-based test data generation [1] is attracting attention as a method for generating test data from formal specifications. In this approach, the specification is converted into an equivalent expression called *functional scenario form* (FSF). The FSF is a disjunction of multiple independent functional scenarios and each functional scenario (FS) is a conjunction of *test condition* and *defining condition* expressed in a mathematical expression. The test condition only involves input variables of the operation while the defining condition must involve some output variables. when the test condition holds on the input variables, the output variables will be defined by the defining condition. Currently, test data generation from a functional scenario only takes the test condition into account and leaves the defining condition untouched [2-4]. Thus, the code implementing the defining condition, which can be long and complex, may not be thoroughly tested and bugs existing inside it may not be easily covered. Even if we can use the defining condition for test data generation, the effectiveness of the generated test data from the original defining condition in terms of identifying bugs in the code implementing the defining condition can be extremely limited.

In order to solve this problem, we have used Structured-Objective-based-formal Language (SOFL) [5] as the formal notation for specifications and proposed "Specification-based test case generation with genetic algorithm [6]". The method is characterized by the combination of functional scenario-based test data generation with genetic algorithm and suitable for regression testing in particular. We conducted a case study using two types of test questions to evaluate the proposed method. The results showed that it has a possibility to be useful for generating test data for killing more program mutants than traditional methods, especially for complex functional scenarios. On the other hand, the proposed method can only work on arithmetical relationships between inputs and outputs in which outputs affect the generation of inputs. Therefore, in this paper, we propose specification-level constrained operations using genetic programming as one means to enable more flexible operations, and discuss the effects of the proposed ideas.

The remainder of this paper is organized as follows. Section 2 presents the background of our research and related works, Section 3 proposes the specification-level constrained operations using genetic programming. Section 4 then discusses the effectiveness from the viewpoint of clarity of chromosome notation and ability to search for solutions and Section 5 concludes the paper.

II. Related Work

A. Background

A number of related studies have already been reported on formal specification-based testing (SBT). For example, Mahmood puts together a thesis [7] which is a systemtic review on researches concerning Automated Test Data Generation (ATDG) technology during the period 1997-2006. Offutt and Liu [2] has reported a technique that can be used for automated test data generation from SOFL specification. The technique basically addresses the issue of developing formalizable and measurable criteria for generating test cases from specifications. Khurshid and Marinov report on TestEra [8], a framework for automated specification-based testing of Java programs. TestEra requires as input a Java method (in sourcecode or bytecode), a formal specification of the pre- and postconditions of that method, and bound that limit the size of the test cases to be generated. Using the method's pre-conditions, TestEra will automatically generate all non-isomorphic test inputs up to the given bound. Martins et al. has reported ConData [9], a test auto-generation method for communication protocols specified as extended finite state machines. It is a test generation method that combines various specification-based test methods such as

978-1-7281-8915-4/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS-C51114.2020.00027

transition testing for the control part of a protocol, and syntax and equivalence partitioning for the data part.

On the other hand, a method using a probabilistic search method such as a genetic algorithm has also been proposed. The technique proposed by Pargas et al. [10] is used to automatically search for test data using genetic algorithms with control dependence graphs of the program. In this method, the genetic algorithm conducts its search by constructing new test data from previously generated test data that are evaluated as good candidates. Harman et al. introduced a mutation-based test data generation approach [11], which targets strong mutation adequacy and is capable of killing both first and higher order mutants. Madronal et al. have proposed a method [12] that combines stochastic mutation and random selection. These studies are just examples, and research on software testing using evolutionary computing is becoming active, with special sessions being planned at major international conferences on evolutionary computing such as IEEE Congress on Evolutionary Computation.

All the above techniques of SBT deal with developing specifications for test data generation or directly generating the test data from the existing specifications. However, it may not be possible to easily find out the bugs present in the program code by only using the relationships of inputs from specifications. To solve this problem, we have proposed a formal specification using SOFL and test case generation using GA, focusing on the relationship between input and output [6].

B. *Specification-based Test Case Generation with Genetic Algorith*

1) Functional Scenario-based Testing: We used SOFL as the formal notation for specifications. Because, SOFL as a formal notation is more comprehensible than other formal notations due to the combination of comprehensible condition data flow diagrams (CDFD) for system structure and pre- and post- conditions for defining individual operations in the system. Another reason is its use in industry has been increasing [13]. The most advanced technique for test data generation from formal specifications is known as functional scenario-based test data generation. Figure 1 outlines the flow of program development and functional scenario-based testing.

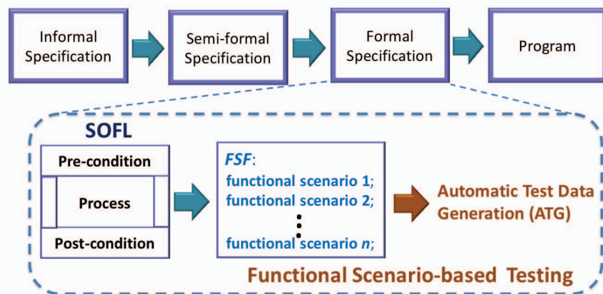

Fig. 1. The flow of program development and functional scenario-based testing.

In this approach, the specification is converted into an equivalent expression called functional scenario form (FSF). FSF is a disjunction of functional scenarios and each functional scenario (FS) is a conjunction of test condition and defining condition.

Definition 1 *An FSF of process* S *is the disjunction of functional scenarios:*

$\vee_{i=1}^{n}(T_i \wedge D_i)$ $(i = 1, \cdots, N)$ *where* $T_i = S_{pre} \wedge G_i$ *is called a* test condition*, which is the conjunction of the pre-condition* Spre *and a* guard condition G_i*, and* D_i *is a predicate called a* defining condition.

The pre-condition S_{pre} of process S is a constraint on the input and it contains only input variables. A guard condition G_i is part of the post-condition but contains no output variables. A defining condition D_i is also part of the post-condition but contains at least one output variable. The functional scenario $T_i \wedge D_i$ describes a single specific functional behavior: when test condition T_i is true, the output of the operation is defined using defining condition D_i. In this paper, we assume that any FSF $\vee_{i=1}^{n}(T_i \wedge D_i)$ of process S is complete, which means that any input satisfying S_{pre} must make $\vee_{i=1}^{n} T_i$ true. An example of Process Mod for finding the quotient q and remainder r from dividing y by x is shown in Figure 2.

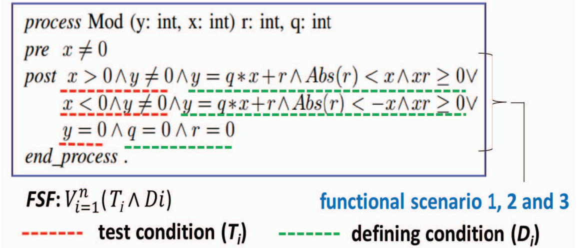

Fig. 2. An example of Process Mod for finding the quotient q and remainder r from dividing y by x.

2) Test Case Generation using GA: In the case of a complex program, it is difficult to directly generate the inputs that satisfies the definition condition without knowing the outputs. For instance, suppose input variable x and output variable y satisfy the defining condition "$x * y > x + y$", we cannot generate input x from "$x * y > x + y$" for the unknown output y. Therefore, usually "$x * y > x + y$" is not used to help generate the input, but used to check the result of executing the program with input x. However, by assigning good values to output variables, we can get some useful reformed specifications. For the defining condition $x * y > x + y$ mentioned above, input data generated from $x * 3 > x + 3$ (when $y = 3$) may be more likely to trigger bugs than that of $x * 1 > x + 1$ (when $y = 1$). In this way, the reformed specifications that keeps the constraints of only input variables can be directly used for test data generations. To obtain this kind of useful reformed specifications, we apply GA for seeking good values for outputs from the defining condition.

Figure 3 shows an overview of the specification-based test case generation system using GA. We defined each functional scenario as a phenotype chromosome and applied modified GA to generate reformed specifications. Specification mutants are first created and then GA is used to find the best mutant test condition and test data is generated from the mutant test condition. Thea expected effect of the test data generated in this

way is to find more bugs in the code. Here, Mutant testing, also called program mutation [14], is used to design test cases and evaluate the quality of existing testing techniques. In mutation testing, some small modifications are injected into the original program. Each mutated version is called program mutant and a test case is regarded as good one for it killing program mutants by making the behaviors of program mutants different from that of the original program. Using this Mutant testing, we compared the test cases created by the conventional ATG and the test cases generated by the proposed method, and showed the effectiveness of the proposed method [6].

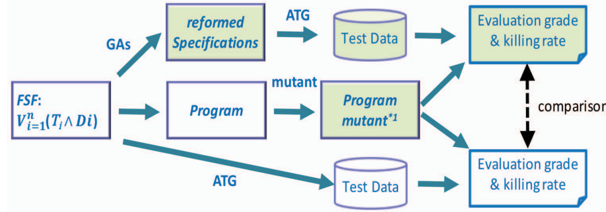

Fig. 3. An overview of the specification-based test case generation system using GA.

We defined a genotype chromosome is a vector constructed by output variab;es and dummy variables: $o' = (o_1, \ldots, o_n, d_1, \ldots, d_c)$, where o_i ($i = 1, \ldots, n$) are output variables, and d_i ($i = 1, \ldots, c$) are dummy variables. That is, for an equation f(inputs, outputs) = 0 from any defining condition D_i, modify it into an inequality $d_1 \leq f$(inputs, outputs) $\leq d_2$. Below is an example of the correspondence between phenotype and genotype chromosomes.

Phenotype chromosome:

$$x > 0 \wedge y \neq 0 \wedge d_1 \leq q * x + r - y \leq d_2 \wedge \mathrm{Abs}(r) < x \wedge xr \geq 0$$

Genotype chromosome: (q, r, d_1, d_2)

For crossover operation, a pair of individuals $[T_i \wedge D_i]_{o'1}$ and $[T_i \wedge D_i]_{o'2}$ from the current population are selected as parents and get their corresponding genotype chromosome o'_1 and o'_2, then, between the two genotype chromosomes, the value of each locus is swapped by the probability of the crossover rate p_c. In the mutation, the value of each locus on the genotype chromosome after crossover changes slightly with a probability of mutation rate p_m.

Here, parents is selected according to the evaluation function Grade. This function is to evaluate an individual or a solution $[T_i \wedge D_i]_{o'}$ by assigning a fitness value. let *Datas* = data_suit_from($[T_i \wedge D_i]_{o'}$) which is a data suit from the individual$[T_i \wedge D_i]_{o'}$, let $N_kill_{i,o'} = (k_1, \ldots, k_m)$ where k_j is the number of datas from $[T_i \wedge D_i]_{o'}$that is able to kill the program mutant mu_j. A test case kills a program mutant means that this test data fails based on the original specifications after it is executed by the program mutant. We consider both the killing rate of program mutants and killing rate of a data suit, so the grade for $[T_i \wedge D_i]_{o'}$ is calculated as:

$$Grade([T_i \wedge D_i]_{o'}) = \frac{Rate_kill(N_kill_{i,o'}) \cdot Sum(N_kill_{i,o'})}{(m \cdot (length(Datas) + 1))}$$

$$\text{where} \begin{cases} Rate_kill(N_kill_{i,o'}) = \frac{\sum_j^m I(k_j>0)}{length(N_kill_{i,o'})} \\ I(k_j > 0) = \begin{cases} 1 & k_j > 0 \\ 0 & k_j \leq 0 \end{cases} \end{cases}$$

We evaluate all individuals using the Grade function, sort them in descending order, then weed out the bottom 50% of individuals, select the parent individual based on the remaining 50%, and apply crossover and mutation operation. To supplement the deleted individuals to form a new population for the next generation.

III. Test Case Generation using Genetic Programming

The results of case studies of our proposed method presented in the previous section showed that, for complicated function scenarios, the method efficiently generates useful test data for killing as many as possible program mutants [6]. But, there were also some limitations that the proposed method can only work on arithmetical relationships between inputs and outputs in which outputs affect the generation of inputs. Here, we propose ideas to improve the operability and the accuracy of solution search by using genetic programming (GP) [15] with restrictions on genetic manipulations.

A. Overview of genetic programming

In GP, a LISP program (symbolic expression: S expression) having the following features is an individual.

- The only target handled by LISP is a symbolic expression "S expression".
- S-expression consists of atoms or lists.
- Atom is a symbol such as a number such as 0, 3.14, or a character string such as X, Y, AND.
- A list is recursively defined as an arbitrary number of atoms including 0 and a list enclosed by left parentheses (and right parentheses). For example, (), (1 2 4), (AND (X Y)), etc.
- LISP receives an S-expression as input and outputs the S-expression as the result of evaluating it. Thus, the LISP program is an S-expression.
- When an S-expression of the form $(F\, x_1\, x_2 \ldots x_n)$ is input, $x_1\, x_2 \ldots x_n$ is evaluated first, and then the function F whose argument is the result is evaluated.

Therefore, for example, an S-expression (LISP program) that realizes (3+1) ∗2 is expressed as (∗ (+3 1) 2). In GP, it is necessary to design the function used in the S-expression and the type and range of the atom in advance.

Genetic manipulation in GP is mutation, crossover and Inversion. Mutation operation is defined by changing atoms in an individual, crossover is exchange of lists between different individuals, and inversion is exchange of sub-lists in an individual. Figure 4 shows an example of these three types of operations.

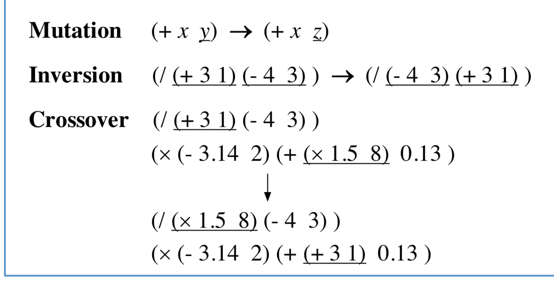

Fig. 4. An example of mutation, inversion and crossover operations in GP.

B. Constrained GP for SBT

Since genetic manipulation based on FSF, which is a mathematically described specification, is easy and it is easier to define genetic manipulations that search for a wider range of solutions than GA, we consider SBT using GP here.

The Phenotype chromosome exemplified in the previous section is expressed below in GP. The first underlined sub-list is the test condition and the remaining sub-lists are the defining condition.

(∧ (∧ (> (*x* 0)) (≠ (*y* 0))) (> (- (+(∗ *q x*) *r*) y) d_1) (< (- (+(∗ *q x*) *r*) y) d_2) (< (Abs *r*) *x*) (≥ (∗ *x r*) 0)))

When variables *q*, *r* and dummy variables d_1 and d_2 overlap between sub-lists as in this example, it is appropriate to define a genotype chromosome and perform genetic manipulation as in the previous section. On the other hand, when independent variables are used between sub-lists, this phenotype chromosome can be used as it is for genetic manipulation of GP. In addition, genotype level crossover can be performed between sub-lists with different numbers of atoms and between individuals with different numbers of sub-lists. In the following, we will describe the improved methods for efficiently applying GP to SBT.

1) List attribute definition: There should be no genetic manipulation between the test condition and defining condition sections. Therefore, both have different attributes and different list notations are defined. For example, the list of test condition parts is written in [] instead of ().There is a constraint that genetic operations are performed only within the same list notation. For example, the phenotype chromosome is expressed as follows.

(∧ [∧ (> (*x* 0)) (≠ (*y* 0))] (> (- (+(∗ *q x*) *r*) y) d_1) (< (- (+(∗ *q x*) *r*) y) d_2) (< (Abs *r*) *x*) (≥ (∗ *x r*) 0)))

2) Atoms attribute definition: When only specific variables are to be manipulated, a genetype chromosome consisting of only the variables to be manipulated as described in the previous section may be used separately. However, it is also possible to deal with phenotypic chromosomes by giving each atom an attribute as to whether or not it is targeted for genetic manipulation. For example, the following shows an example in which only the target atom is bolded.

(∧ [∧ (> (*x* 0)) (≠ (*y* 0))] (> (- (+(∗ ***q*** *x*) ***r***) y) $\boldsymbol{d_1}$) (< (- (+(∗ ***q*** *x*) ***r***) y) $\boldsymbol{d_2}$) (< (Abs ***r***) *x*) (≥ (∗ *x* ***r***) 0)))

Alternatively, a wider range of search may be realized by not limiting the operation to a specific variable. Below is an example of a crossover that exchanges sub-lists on a phenotype chromosome with different gene length of the defining condition part. That is, it is possible to exchange sub-lists between different forms of function definitions.

P_1: (= (+ (∗ *a x x*) (∗ *b x*) *c*) y)

P_2: (= (+ (∗ *d x*) *e*) y)

→

P_1': (= (+ (∗ *a x x*) (+ (∗ *d x*) *e*) *c*) y)

P_2': (= (∗ *b x*) y)

3) Knowledge database using ADF: GP has a function called Automatically Defined Functions (ADF) [16] that defines frequently used lists as subroutines to improve the efficiency of searches and simplify chromosome expressions. By utilizing this ADF function as a knowledge database and registering the FSF or list that was effective for bug detection as an ADF, it is possible to efficiently search for test cases that are effective for bug detection.

4) Visualization: Another advantage of using GP is that it allows visualization of the chromosome, the FSF during the search. The LISP S-expression can be represented by a tree structure, so by applying the S-expression to the FSF notation, the tree structure can be used to visualize the FSF of the search process. Figure 5 shows the tree structure of the defining condition part, taking the crossover using P1 and P2 as an example.

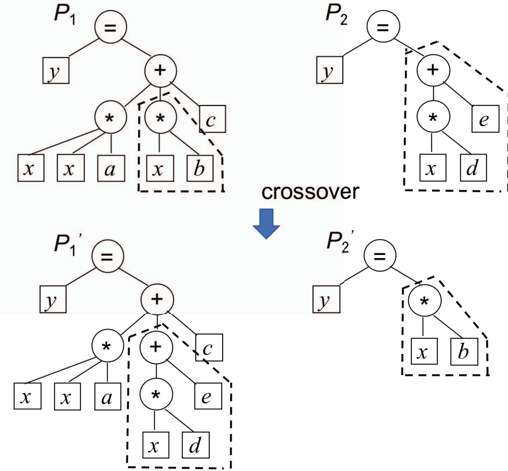

Fig. 5. An example of crossover operation using tree structure representation.

IV. Discussion

A. Gene notation and solution search ability

Introducing GP with some restrictions on notation and genetic manipulation makes it possible to represent chromosomes (ie, FSFs) in a format that conforms to the LISP language. This allows a mathematically described specification

(FSF) to be directly defined as a gene, allowing FSF to be regenerated from a chromosome. In addition, it is possible to perform genetic manipulation between chromosomes having different gene lengths, that is, between functions having different orders, and it is expected that the solution search ability is improved. Figures 6 and 7 show the experimental results of the previous report [6] using GA. Basically, the method using GA succeeds in detecting more mutants than the conventional method, but in some cases, the same number of mutants as in the conventional method can be detected.

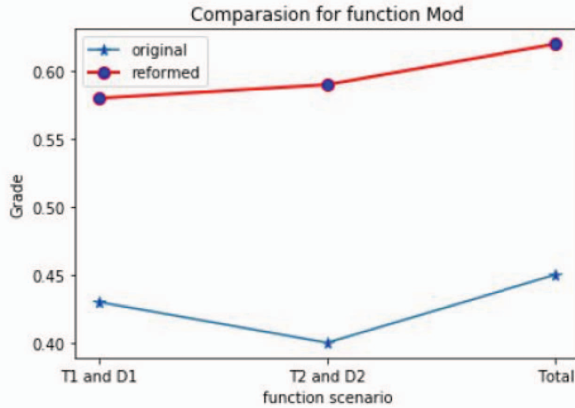

Fig. 6. Comparison of the ability to detect mutants in the Mod function between the conventional method and the test case generation method using GA.

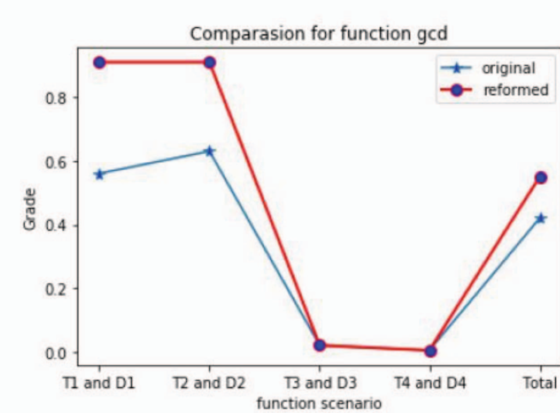

Fig. 7. Comparison of the ability to detect mutants in the gcd function between the conventional method and the test case generation method using GA.

One possible cause is that GA restricts gene manipulation only between chromosomes with the same gene length. Therefore, it may be improved by the use of GP, which allows genetic manipulation between different gene lengths. However, in the future, more rigorous comparative evaluation will be required, such as in computer simulations.

B. Possibility of learning efficiency improvement using knowledge database

Since GA and GP are probabilistic search methods, it is possible to search using the reinforcement learning function even for problems for which the objective function is not known explicitly. On the other hand, it is expected that the search ability will be improved by using the knowledge acquired in the past together. Therefore, if the partial list (that is, a part of the FSF) obtained in the past and effective for detecting the mutant can be registered and used in the knowledge database, the efficiency of the solution search can be expected to be improved. Here, the GP has a function that can be used for searching by defining a subroutine as an ADF. Figure 8 shows an example of S-expression using ADF. In the figure, PROGN is composed of a main part that determines the value of the entire S-expression (subtree on the right) and a function part (a subtree on the left) that is referred to as a subroutine in the S-expression. ADF0 defined as a function part can be called freely within an S-expression. By defining the partial list obtained in the past that was effective for mutant detection as an ADF function at any time, a search using a knowledge database can be easily realized. It is considered that the solution search efficiency is improved as it is used, and the expression of chromosomes can be simplified.

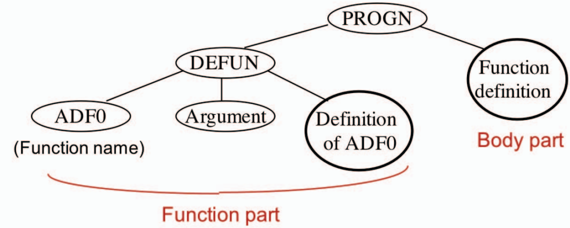

Fig. 8. An example of S-expression using ADF.

C. Visualization for solution search result analysis

GP defines a chromosome with the LISP S expression. Since the LISP S-expression can be represented by a tree structure, we can use the tree structure to visualize the FSF of the search process by applying the S-expression to the FSF notation. It is possible to use this visualization function for FSF analysis that is effective in detecting mutants. In addition, by defining the FSF that is found as a result of the analysis and is effective for detecting mutants as an ADF, it can be used for solution search from the next time. Visualization of the tree structure is considered to be effective as an effective FSF analysis method for detecting mutants, and as a method for selecting ADF definition candidates that contribute to efficient solution search. However, as a future issue, it is necessary to verify the effectiveness through concrete evaluation experiments.

V. CONCLUSIONS

We have already proposed a test data generation method based on functional scenarios using GA, mainly for regression tests. This method obtains a reconstructed specification that is useful for bug detection, sensitive to small syntactical changes in program code, by assigning appropriate values to unknown output variables and making changes to the original specification. Therefore, this method mainly deals with unknown output from the formal specification and can generate input data to test the target program. As a result of a comparison experiment with a conventional method of generating a test case from a functional scenario, it tends to generate more useful test data for killing a program mutant, but in some test problems, it was similar to the conventional method. In this paper, as an idea

to improve the operability and accuracy of the solution search of this method, we proposed a combination of specification level constraint operation using genetic programming and knowledge database using automatic definition function. Moreover, we have examined on the desk about the effectiveness of the proposed idea. As a result, we have showed that the idea has a possibility to improve detectability of mutants and could be used for FSF analysis, which is effective for mutant detection. However, a quantitative evaluation experiment by computer simulation will be required in the future.

REFERENCES

[1] S. Liu and S. Nakajima. Combining Specification Testing, Correctness Proof, and Inspection for Program Verification in Practice. In *Proceedings of the 3rd International Workshop on SOFL + MSVL (SOFL+MSVL 2013)*, pages 3–16, Queenstown, New Zealand, October 29 2013. LNCS 8332, Springer.

[2] A. J. Offutt and S. Liu. Generating Test Data from SOFL Specifications. *Journal of Systems and Software*, 49(1):49–62, December 1999.

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

[4] Y. Sato and T. Sugihara. Automatic generation of specification- based test cases by applying genetic algorithms in reinforcement learning. In *International Workshop on Structured Object-Oriented Formal Language and Method*, pages 59–71. Springer, 2015.

[5] S. Liu. *Formal Engineering for Industrial Software Development: Using the SOFL Method*. Springer Science & Business Media, 2013.

[6] R. Wang, Y. Sato and S. Liu. Specification-based Test Case Generation with Genetic Algorithm. In *Proceedings of the 2019 IEEE Congress on Evolutionary Computation.* pp. 1359-1366, IEEE, 2019.

[7] Shahid Mahmood. A systematic review of automated test data generation techniques, 2007.

[8] Sarfraz Khurshid and Darko Marinov. Testera: Specification-based testing of java programs using sat. *Automated Software Engineering*, 11(4):403–434, 2004.

[9] Eliane Martins, Selma B Sabia˜o, and Ana Maria Ambrosio. Condata: a tool for automating specification-based test case generation for communication systems. *Software Quality Journal*, 8(4):303–320, 1999.

[10] Roy P Pargas, Mary Jean Harrold, and Robert R Peck. Test-data generation using genetic algorithms. *Software testing, verification and reliability*, 9(4):263–282, 1999.

[11] Mark Harman, Yue Jia, and William B Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 212–222. ACM, 2011.

[12] Lorena Gutierrez-Madronal, Antonio Garcia-Dominguez and Inmaculada Medina-Bulo. Combining Evolutionary Mutation Testing with Random Selection. In *Proceedings of the 2020 IEEE Congress on Evolutionary Computation.* pp. 1-8 (CD-ROM), IEEE, 2020.

[13] Juan Luo, Shaoying Liu, Yanqin Wang, and Tingliang Zhou. Applying sofl to a railway interlocking system in industry. In *International Workshop on Structured Object-Oriented Formal Language and Method*, pages 160–177. Springer, 2016.

[14] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[15] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of* Natural Selection. Cambridge, MA: The MIT Press, 1992.

[16] John R. Koz. Genetic Programming II: Automatic Discovery of Reusable Programs. Cambridge, MA: The MIT Press, 1994.