mathematics

MDPI

Article

Mutated Specification-Based Test Data Generation with a Genetic Algorithm †

Rong Wang [1,*], Yuji Sato [1] and Shaoying Liu [2]

[1] Department of Computer Science, Hosei University, Tokyo 184-8584, Japan; yuji@hosei.ac.jp
[2] Graduate School of Advanced Science and Engineering, Hiroshima University, Hiroshima 739-8511, Japan; sliu@hiroshima-u.ac.jp
* Correspondence: rong.wang.99@stu.hosei.ac.jp
† This paper is an extended version of our paper published in 2019 IEEE Congress on Evolutionary Computation (CEC), Wellington, New Zealand, 10–13 June 2019.

Abstract: Specification-based testing methods generate test data without the knowledge of the structure of the program. However, the quality of these test data are not well ensured to detect bugs when non-functional changes are introduced to the program. To generate test data effectively, we propose a new method that combines formal specifications with the genetic algorithm (GA). In this method, formal specifications are reformed by GA in order to be used to generate input values that can kill as many mutants of the target program as possible. Two classic examples are presented to demonstrate how the method works. The result shows that the proposed method can help effectively generate test cases to kill the program mutants, which contributes to the further maintenance of software.

Keywords: test data generation; genetic algorithm; specification-based testing; regression testing; mutation testing

Citation: Wang, R.; Sato, Y.; Liu, S. Mutated Specification-Based Test Data Generation with a Genetic Algorithm. *Mathematics* **2021**, *9*, 331. https://doi.org/10.3390/math9040331

Academic Editor: David Greiner
Received: 31 December 2020
Accepted: 4 February 2021
Published: 7 February 2021

1. Introduction

Regression testing is an important technique to ensure that previously tested software still performs in the same way after it is changed or integrated with other software [1–3]. In general, changes to software are mainly concerned with the efficiency enhancement, robustness improvement, and configuration changes, but these changes should not result in big alternation of the functionality defined in the specification of the software. Therefore, specification-based testing (SBT) methods can be effectively used in regression testing.

SBT is characterized by test data being generated from the specification without concerning the structure of the corresponding program and test results being analyzed based on the specification [4–8]. Formal specifications may allow the test data generation and test result analysis to be done rigorously, systematically, and even automatically in many cases [9–12]. In our work, we mainly use formal specifications in pre- and post-conditions, such as Vienna Development Method (VDM) [13], a formal method that has been developed over past years [14,15], and Structured-Object-Oriented Formal Language (SOFL) [16], which has the potential of practical use in industry and serves as a solid foundation to develop a method of *functional scenario-based test data generation* [17].

However, in spite of considerable progresses having been made, it is still not easy for SBT to generate various test data only from specifications to detect different bugs that are contained in the program. This is because different features and effects of the program output cannot be controlled and triggered by only input data suites that satisfy some parts of the specification (some constraints over only input variables). Consequently, many faulty program paths would not be detected and thus the bug detection would be likely to fail in some cases. For the existing SBT, one of the major deficiencies is that the test data generation only considers the constraints over input variables in the formal specification, without making use of the constraints over outputs before the execution of a program.

To overcome the shortcomings of the existing SBT methods, we propose a new method for test data generation in this paper. The proposed method introduces dummy variables into some specified constraints in the specification, and makes use of the constraints over both input and output variables to guide the test data generation, in contrast to the conventional SBT methods that concerns only constraints over input variables. This method features the combination of three techniques, SBT, mutation testing, and GA. It is to obtain the enhanced (mutated) formal specifications by using a GA so that input data generated from these specifications are more likely to kill different kinds of mutants of the target program under test. The expected effect of the test data generated in this way is to detect various bugs probably occurring in the program that is being developed or improved.

The work in this paper is an extension for our previous work in [18]. Comparing with the previous work, we use more different kinds of mutants (16 types against previously 10) for each program in our case study to gain better mutated functional scenarios that are capable of detecting more bugs. In addition, we give additional experimental results of the proposed approach that uses no dummy variables, and point out the importance of introducing dummy variables. In addition, we carefully conduct more experiments with the dummy variables that are introduced to reform different parts of original specification, the equality and inequality relations. Based on that, we enriched the analysis for the effect of different ways of introducing dummy variables on bug detection.

The remaining part of the paper is organized as follows. Section 2 briefly introduces the existing work related to our approach. Section 3 illustrates how to transform formal specifications to chromosomes as well as the corresponding genetic manipulations in GA. Section 4 describes the main procedures of obtaining desirable reformed specifications for test data generation by integrating GA. Section 5 presents two classic cases to demonstrate the feasibility and efficiency of the approach. Finally, Section 6 concludes the paper and points out future research directions.

2. Related Work

In this section, we introduce several advanced techniques that relate to our methodology for test data generation.

Data flow analysis, a technique for computing the def-use associations for the control flow graph (CFG) of a program, are often used to develop different strategies for test data generation over a long history [19–21]. Many research works have proposed promising methods for automatic test data generation that integrates the data flow analysis with the heuristics, such as GAs [22–24], particle swarm optimization [25], and ant colony optimization [26]. Different from these techniques, our approach conducts the testing under the circumstance where the source code of thirty-party library under test cannot be accessed. Thus, these techniques rely too much on the knowledge and analysis of internal design (or code structure), but our approach focuses on generating test data without analyzing the source code when applying the GA to the formal specification. In addition, with respect to the usage of the GA, our approach uses the GA to search for the optimal mutated specifications that are later used for input data generation, while the techniques mentioned above use the GA to directly search for good input data.

The techniques of mutation-based test data generation [27,28] are used to select a set of "good" test data by executing designed incorrect versions of an original program with a great number of test data from the domain. Test data are selected if it can cause unintended behaviors for a certain number of incorrect versions. These techniques mainly concentrate on designing appropriate mutation operators to introduce small modifications for different kinds of programming languages such as Java [29], C# [30], and C++ [31]. The incorrect versions, also called program mutants, are created by inserting the mutant operators into the original program. Compared with these techniques, our method selects a set of "good" mutated specifications as a seed for further test data generation by using not only program mutants but also the mutated specifications with the GA.

The SBT techniques, some of them integrated with heuristic search strategies, have been well developed to cope with different kinds of specifications, such as SOFL [9,16], Alloy [32,33], protocol specifications [34,35], and Object Constraint Language (OCL) specification [36,37]. Among these specifications, we take an interest in the formal specification of pre-post style like SOFL and Alloy. On the one hand, the SBT techniques for both SOFL and Alloy generate test data only from the pre-condition and use the post-condition as an oracle to check if the outcome is correct. On the other hand, the SBT with SOFL still needs to be improved since a data suite generated only from the original SOFL specification is not sufficient enough to trigger different kinds of bad behaviors of programs. On the contrary, our approach uses both the pre- and post- conditions to generate input data, as well as selects the optimal mutated specifications to enhance the bug detection.

3. GA with Mutated Specification

We first briefly introduce the basics of GA and then discuss how to obtain reformed specifications.

3.1. Description for GA

GA is a heuristic search method inspired from evolutionary biology and was first proposed by John Holland [38]. In general, a GA is involved in an iterative process with three steps: (i) create an initial population of solutions (called individuals) represented by a pre-defined chromosome that are typically encoded the solutions to a problem; (ii) in the existing population, select a group of individuals by a specified *selection* strategy based on a *fitness* function, and generate the next population from applying two key genetic operators, *crossover* and *mutation* to those selected individuals; (iii) repeat step (ii) until the remaining individuals in the generation are good enough according to both the fitness function and the stop criteria.

Since GA works well in finding optimal solutions for nonlinear problems and the specifications of pre-post style could be easily transformed to chromosomes with few efforts, we employ GA to find the best mutated specifications in this paper. Later, we will first describe how to transform the original formal specifications into a chromosome, and then carefully describe the evolution in step (ii) in detail for our specific goal: to obtain all the mutated functional scenarios from the specification, each a constraint over only input variables.

3.2. Mutation Testing

Mutation testing, also called program mutation [39], is used to design test cases and evaluates 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 test data are regarded as the good one if it kills the program mutants, that is, it makes the behavior of program mutants different from that of the original program.

In our approach, both programs and the specifications are mutated. The program mutants are used to evaluate the quality of mutated specifications. We search for good mutated specifications that can be used to effectively generate test data for bug detection.

3.3. Mutated Specifications

We use SOFL as the formal notation for specifications in this paper. There are two reasons: one is that SOFL, as a formal notation, is more comprehensible than other formal notations since it uses the comprehensible condition data flow diagrams for system structure as well as pre- and post- conditions for defining individual operations in the system. Another reason is that SOFL is familiar to us and its use in industry has been increasing [40].

In SOFL, the *defining condition* describes the constraints over input and output variables after a method in the program performs. Generally, the defining condition is not used for directly generating input data in most of existing techniques because the values of outputs

in defining condition are unknown to us before the execution of the program. We consider the defining conditions as an important factor for test data generation from which the test data are sensitive to bad behaviors of the program.

Since defining conditions describe how output variables relate to input variables, they are often used to check whether an execution of the program is correct or not, rather than being used to directly generate input values. For a program, it is usually difficult to directly generate input values that satisfy a defining condition without knowing the corresponding output values. For instance, suppose input variable x and output variable y satisfy the defining condition $(x * y > x + y)$, we cannot generate input x from $(x * y > x + y)$ due to the unknown output y. Thus, usually $(x * y > x + y)$ is not used to help generate the input values but can be used to check the result of executing the program with input x.

Nevertheless, by assigning appropriate values to the output variables in the defining conditions, we can get some useful mutated specifications that can be used to directly generate input values. For the defining condition $(x * y > x + y)$ mentioned above, input data generated from $(x * 2 > x + 2)$ (when $y = 2$) may be more likely to trigger bugs than from $(0 > x)$ (when $y = 0$). Currently, it cannot be determined without further checking. However, $(x * 2 > x + 2)$ is definitely better than $(x * 1 > x + 1)$ (when $y = 1$) because the latter is always false and cannot be used to generate test data.

Our work mainly concentrates on developing a way to find appropriate output values for the specification. These output values are then used to build the mutated specifications that are the constraints over only input variables. Then, the mutated specifications can be directly used to generate input values in regression testing. To achieve that, we employ GA to seek such appropriate values of outputs from the defining condition.

Moreover, to obtain mutated specifications that are more powerful in bug detection, some extension is considered for reforming defining conditions before applying GA. In this extension, we make a slight change in defining conditions so as to induce the generated test data that satisfy those reformed ones to trigger as many bad behaviors of the program as possible.

In our method, *mutated specifications* are made from after applying GA to the original specification. More precisely, the mutated specifications can be obtained by following *two rules*:

1. Reforming the original specification by introducing dummy variables into defining conditions so that test data that satisfy those reformed ones can trigger bad behaviors of the program;
2. Finding appropriate concrete values through GA and assigning them to the output and dummy variables that occur in the reformed specification.

Our goal is to obtain a new version of the specification from which the test data suite can be generated to trigger as many bugs as possible in the program. Next, we will define the chromosome forms for the reformed specification, as well as describe the crossover operator and mutation operator. Then, we apply the GA for gaining the suitable mutated specifications that can do well in bug detection.

We define the form of chromosomes for a condition data flow diagram (CDFD) that is part of the SOFL language.

A condition data flow diagram (CDFD) is a directed graph that specifies how processes work together to provide functional behaviors [41]. Every process has its own pre- and post-conditions. For instance, Figure 1 displays a small CDFD that consists of two processes A and B where process A first consumes two input variables x and y and produces output z, and then process B consumes z and produces w.

The two separately defined processes A and B may not be automatically combined into a bigger process C since we can not always infer $C\_pre(x, y) \wedge C\_post(x, y, w)$ just from $A\_pre(x, y) \wedge A\_post(x, y, z) \wedge B\_pre(z) \wedge B\_post(z, w)$ unless we know the expression $z = Expr(x, y)$ in $A\_post(x, y, z)$, since, in that case, we can easily replace z with $Expr(x, y)$ and derive the following predicate expression:

$$C\_pre(x,y) \wedge C\_post(x,y,w) = A\_pre(x,y) \wedge A\_post(x,y,Expr(x,y)) \wedge B\_pre(Expr(x,y)) \wedge B\_post(Expr(x,y),w).$$

However, the intermediate variables between two processes like variable z can not always be replaced in real CDFDs. Therefore, our discussion on test data generation from specifications focuses on a single process.

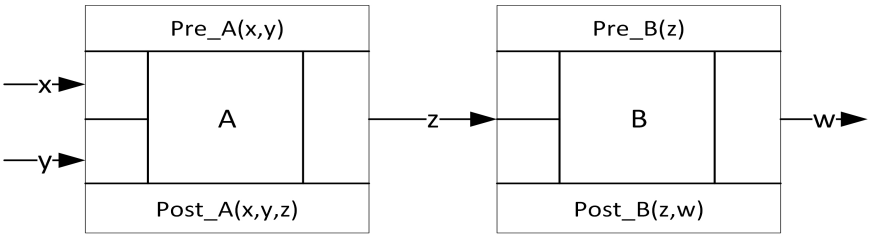

Figure 1. The process A and process B.

3.4. Chromosome Formation

In this approach, the specification is converted into an equivalent expression called *functional scenario form* (FSF).

Definition 1. *A FSF of process is the disjunction of functional scenarios:* $\vee_{i=1}^{n}(T_i \wedge D_i) := S_{pre} \wedge (\vee_{i=1}^{n}(G_i \wedge D_i))(i = 1, \cdots, N)$ *where* $T_i = S_{pre} \wedge G_i$ *is called a test condition, which is the conjunction of the pre-condition* S_{pre} *and the 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.

Each functional scenario defines an independent function of the operation: 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 meanwhile leaving the defining condition untouched [9,42,43].

Now, we explain how to make a slight extension to change the form of defining conditions so as to allow bad behaviors to occur. To obtain a more flexible and useful reformed specification, we introduce *dummy variables*, $d_i(i = 1, \cdots, c)$, to the relationship of inputs and outputs from the defining condition. Then, we build an *output vector* from both the *dummy variables* and output variables.

Definition 2. *An output vector* $o^{'}$ *is a vector constructed by output variables and dummy variables:* $o^{'} = (o_{1,} \cdots, o_n, d_{1,} \cdots, d_c)$*, where* o_i $(i = 1, \cdots, n)$ *are output variables, and* d_i $(i = 1, \cdots, c)$ *are dummy integer variables.*

For a relation $(f(inputs, outputs) \triangle 0)$ (where $\triangle$ is a operator such as $=, >, <$ ...) in the defining condition D_i, by introducing dummy variables d_1 and d_2, we construct an inequality $d_1 <= f(inputs, outputs) <= d_2$ and replace the relation f with this new inequality in D_i. Then, the output vector is formed as $o^{'} = (o_{1,} \cdots, o_n, d_1, d_2)$. In our work, we mainly make such change to only equality relations.

We change an equality relation to such an inequality relation because an equality relation is quite a strict condition that would drastically narrow down the exploration of input values for a single functional scenario when output values are determined by GA.

Therefore, dummy variables need to be introduced for equality. For the inequality relation in the specification, dummy variables are not introduced to them because, compared with equality relation, inequality relation is not a too strict condition for the generation of input values. Thus, these kind of relations are used to preserve some original features of specifications. In addition, the experimental results in Section 5 also indicate that additional dummy variables for inequality cannot help considerably improve the quality of the mutated specifications.

Definition 3. *A chromosome* $[T_i \wedge D_i]_{o'}$ $(i = 1, \cdots, N)$ *is a reformed functional scenario* $T_i \wedge D_i$*, where some dummy variables are introduced to* D_i*. An individual (a mutated specification) is a constraint over symbolic inputs, established from the chromosome* $[T_i \wedge D_i]_{o'}$ *by assigning concrete values to the output vector* $o^{'} = (o_{1,} \cdots, o_n, d_{1,} \cdots, d_c)$*. A population is a group of such individuals. For convenience, the output vector* $o^{'}$ *is also called d-chromo, and each element of* $o^{'}$ *is called a genetic.*

From this definition, a d-chromo $o^{'}$ with concrete values determines an individual formula $[T_i \wedge D_i]_{o'}$ that is a constraint on symbolic inputs. Such an individual is a reformed specification that can be used to generate test data for the program. In order to obtain good individuals to generate test data that are useful for bug detection, we apply the genetic manipulation to a group of individuals $[T_i \wedge D_i]_{o'}$ and find the appropriate d-chromo $o^{'}$. Each individual will be scored by evaluating the quality of the test data that are generated from it.

3.5. Genetic Manipulations and Selection

The genetic manipulation refers to the change of genetic structure in biology, but, in the GA, it indicates that a "child" solution is produced from a pair of "parent" solutions by using genetic operators like crossover and mutation.

In the existing population, a pair of individuals (solutions) are selected as parents to perform the *crossover* operator to produce their offspring. More specifically, as illustrated in Figure 2, first select two individuals $[T_i \wedge D_i]_{o_1'}$ and $[T_i \wedge D_i]_{o_2'}$ from the current population as parents and get their d-chromos $o_1^{'}$ and $o_2^{'}$, then swap each two genetics of the two d-chromos with possibility p $(0 < p < 1)$ to obtain two new individuals.

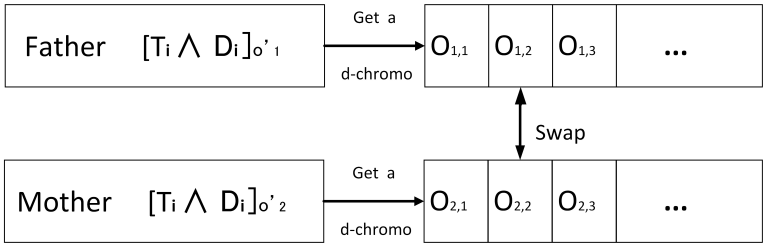

Figure 2. Crossover operator.

To perform the *mutation* operator, each genetic of an individual is mutated with possibility q $(0 < q < 1)$, as displayed in Figure 3. More clearly, for one individual $[T_i \wedge D_i]_{o'}$ with its d-chromo $o^{'} = (o_{1,} \cdots, o_n, d_{1,} \cdots, d_c)$, each genetic of it has the possibility q to be mutated:

$o_i^{'} := o_i^{'} + \triangle$, where $\triangle$ is a different scalar of small value.

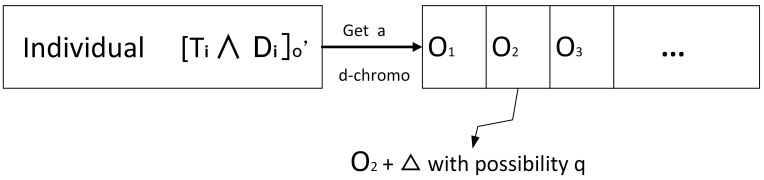

Figure 3. Mutation operator.

Fitness function *Grade* is used to evaluate an individual (a solution) $[T_i \wedge D_i]_{o'}$ by assigning a fitness value. Let $Datas = data\_suite\_from([T_i \wedge D_i]_{o'})$ where $data\_suite\_from$ generates a suite of input data from $[T_i \wedge D_i]_{o'}$ by using a constraint solver. Let $N\_kill_{i,o'} = (k_1, \ldots, k_m)$ where k_j is the number of test data that have been generated from $[T_i \wedge D_i]_{o'}$ and have killed the program mutant mu_j as well. A test case that kills a program mutant indicates that it fails based on the original specifications after it is executed by the program mutant. We consider both the killing rate of program mutants *Kill_rate*, and the killing rate of a data suite as important factors to compute the grade for $[T_i \wedge D_i]_{o'}$:

$$Grade([T_i \wedge D_i]_{o'}) = \frac{Kill\_rate(N\_kill_{i,o'}) \cdot Sum(N\_kill_{i,o'})}{(m \cdot (length(Datas) + 1))} \tag{1}$$

$$where \begin{cases} Kill\_rate(N\_kill_{i,o'}) = \dfrac{\Sigma_{j=1}^{m} I(k_j > 0)}{length(N\_kill_{i,o'})}, \\ I(k_j > 0) = \begin{cases} 1 & k_j > 0 \\ 0 & k_j \leq 0 \end{cases} \end{cases} \tag{2}$$

The factor $\Sigma_{j=1}^{m} I(k_j > 0)$ in *Kill_rate* is intended to encourage each mutated functional scenario to generate a test data suite that can kill as many different kinds of program mutants as possible. The factor $Sum(N\_kill_{i,o'})$ as a part of the numerator in *Grade* would inspire every mutated functional scenario to generate a test data suite where most test data are effective enough to kill as many program mutants as possible.

For a given chromosome $[T_i \wedge D_i]_{o'}$, its individual with appropriate d-chromo $o'_{i,best}$ is regarded as the best if and only if this individual possesses the highest value of *Grade* in the whole population. GA is to find such best individuals for these chromosomes $[T_i \wedge D_i]_{o'}$ $(i = 1, \cdots, N)$.

After all the individuals from the current population are evaluated, GA would select most of the best ones to form a new population for the next generation. This process is called *selection*. In our approach, we evaluate all the individuals and sort them by descending, then select individuals in the top 50 percent of the current population to breed the next generation.

As we can see, GA is used to find the best individuals separately for each chromosomes $[T_i \wedge D_i]_{o'}$ $(i = 1, \cdots, N)$. In order to evaluate all the best individuals represented by different chromosomes, the final formula of evaluation is made as follows:

$$Grade(\vee_{i=1}^{n} [T_i \wedge D_i]_{o'_{i,best}}) = \frac{Kill\_rate(N\_kill) \cdot Sum(N\_kill)}{(m \cdot length(Datas))} \tag{3}$$

$$where \begin{cases} N\_kill = \Sigma_{i=1}^{n} N\_kill_{i,o'_{i,best}}, \\ Datas = \{data\_suite\_from([T_i \wedge D_i]_{o'_{i,best}})\}_i \end{cases} \tag{4}$$

We use the final formula to find all the mutated functional scenarios that together hit the highest final grade (i.e., do the best in bug detection), each mutated one with well-tuned values for dummy variables and output variables. Additionally, this final grade is also used for comparison between our approach and other techniques. In the case study, our

method is compared with the conventional specification-based method with respect to test data generation for bug detection.

4. Algorithm Summary

Our approach that incorporates GA accomplishes the goal of obtaining the mutated specifications by taking three key steps:

1. Inject faults into the original program to obtain a set of program mutants;
2. Use reformed specification $[T_i \wedge D_i]_{o'}$ as seed chromosomes. Each chromosome corresponds to a group of individuals that are generated by assigning concrete values to the output vector in the chromosome;
3. Apply GA to each chromosome and select the best individuals (the best mutated specifications). According to the original specifications, determine whether or not a test case from a mutated specification (a constraint over inputs) kills the program mutants.

Figure 4 displays the whole evolution process of GA. In the first round of evolution, a group of individuals are initialized and evaluated. Then, the best individuals in the top k ($k = 50\%$ in this paper) of the group are selected to perform both crossover and mutation operators to produce their offspring for the next round. In the next round, all of the individuals are evaluated and the top k of them again prepare to breed a new generation by performing genetic manipulations. The population iteratively evolves in this process until there has been no improvement in the population or it reaches the predefined maximum number of generations.

In the mutation testing, we use Z3 [44], a widely used satisfiability modulo theories (SMT) solver, as our constraint solver to generate the data suite for each individual formula (i.e., each mutated functional scenario). The generation for a data suite takes three steps: (1) use Z3 to generate a test data that satisfies the logical formula, (2) exclude all the test data obtained previously from the logical formula; (3) go to step (1) to obtain another piece of test data unless enough test data are obtained. Each individual formula is evaluated by the fitness function that measures the quality of the test data suite.

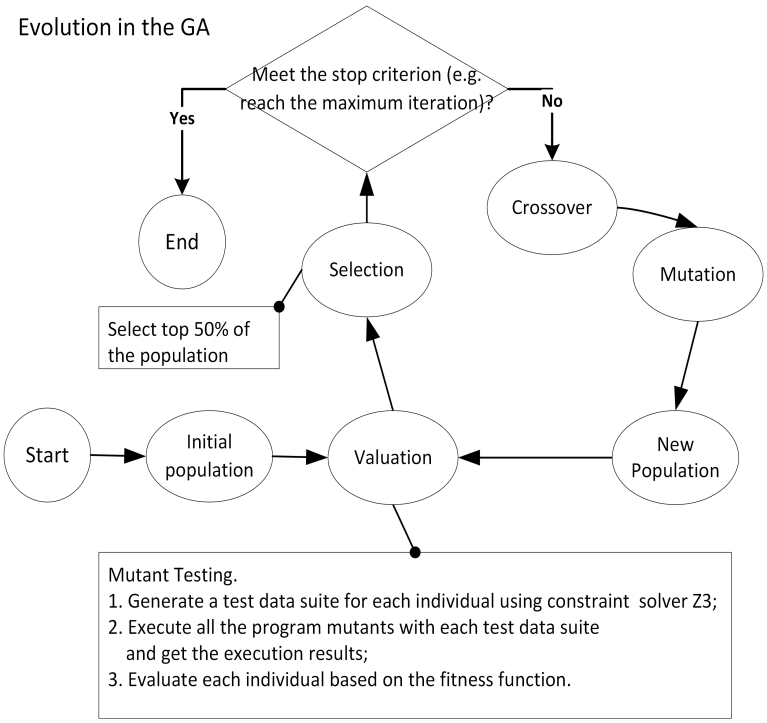

Figure 4. The evolution in GA.

We give the pseudo-code of the algorithm in Appendix A.

5. Case Study

In this section, we apply GA to two classic examples to demonstrate the effectiveness of the proposed method. The original specifications are used as test oracles for determining whether the outputs are correct or not during the evaluation of individuals.

We compare our method with the conventional method, called *original specification-based method*, which directly generates input data from the original specifications by using Z3. In the original specification-based method, neither dummy variables nor defining condition are used to generate input values, since the defining condition contains output variables with unknown values. The input data are directly generated from only test conditions (pre-condition and guard-condition over only input variables) by using Z3.

Sixteen program mutants are prepared for each program in the way that the injected faults will not cause execution crash and infinite loops since we only focus on the functional bugs in this paper. Both methods generate a test suite of the same size 20 every time to execute these program mutants in the evaluation process.

5.1. Case Study 1: Mod

In this program, process *Mod* is to find the quotient q and remainder r from dividing y by x. For *Mod*, we give its formal specification in SOFL and the implementation in Python.

The formal specification of *Mod* is:

process Mod (y: int, x: int) r: int, q: int
pre $x \neq 0$
post $x > 0 \wedge y \neq 0 \wedge y = q * x + r \wedge Abs(r) < x \wedge xr \geq 0 \vee$
$x < 0 \wedge y \neq 0 \wedge y = q * x + r \wedge Abs(r) < -x \wedge xr \geq 0 \vee$
$y = 0 \wedge q = 0 \wedge r = 0$
end_process

Its implementation in Python is:

```
def Abs(x):
if x>=0:
return x
else:
return -x
def mod(y, x):
r = y;
q = 0;
if y!=0:
if x*y > 0:
while Abs(x) <= Abs(r):
r = r - x
q = q + 1

else:
while x*r < 0:
r = r + x
q = q - 1
return r, q
```

In this specification, *Abs* is a function for calculating the absolute value of its input. To shorten the explanation of each step, assume *Abs* is an inline executable predicate. Both $-7\ mod\ 5 = 3$ and $-7\ mod\ 5 = -2$ satisfy the classic definition $y = q * x + r \wedge Abs(r) < Abs(x)$. To avoid the ambiguity, the specification of *Mod* puts an additional condition $xr \geq 0$ in order to get only one result of $-7\ mod\ 5 = 3$.

In the specification, the pre-condition, guard-conditions, and defining conditions are listed as:

$S_{pre} := x \neq 0$;
$G_1 := x > 0 \wedge y \neq 0$; $D_1 := y = q * x + r \wedge Abs(r) < x \wedge xr \geq 0$;
$G_2 := x < 0 \wedge y \neq 0$; $D_2 := y = q * x + r \wedge Abs(r) < -x \wedge xr \geq 0$;
$G_3 := x > 0 \wedge y = 0$; $D_3 := q = 0 \wedge r = 0$.

We can obtain the functional scenarios $T_i \wedge D_i := S_{pre} \wedge G_i \wedge D_i$ as follows:
$T_1 \wedge D_1 := x > 0 \wedge y \neq 0 \wedge y = q * x + r \wedge Abs(r) < x \wedge xr \geq 0$;
$T_2 \wedge D_2 := x < 0 \wedge y \neq 0 \wedge y = q * x + r \wedge Abs(r) < -x \wedge xr \geq 0$;
$T_3 \wedge D_3 := x \neq 0 \wedge y = 0 \wedge q = 0 \wedge r = 0$.

For $T_3 \wedge D_3$, the input x and y are not related to the output q and r, so we do not need to apply GA to it. Since there is an equality $y = q * x + r$ in which inputs and outputs are related, we introduce two dummy variables d_1 and d_2. The chromosomes of *Mod* are displayed in Table 1.

Table 1. Chromosome forms for functional scenarios of process Mod.

Chromosome	D-Chromo	Dummy Vars
$[T_1 \wedge D_1]_{o'} : x > 0 \wedge y \neq 0 \wedge$ $d_1 \leq q * x + r - y \leq d_2 \wedge$ $Abs(r) < x \wedge xr \geq 0$	$o' =$ (q, r, d_1, d_2)	d_1, d_2
$[T_2 \wedge D_2]_{o'} : x < 0 \wedge y \neq 0 \wedge$ $d_1 \leq q * x + r - y \leq d_2 \wedge$ $Abs(r) < -x \wedge xr \geq 0$	$o' =$ (q, r, d_1, d_2)	d_1, d_2

Apply Algorithm A1 to these chromosomes. The results are displayed in Table 2.

Table 2. Results for process Mod after applying GA.

The Best Individual of Chromosome	Grade
$[T_1 \wedge D_1]_{o'}$	
$o' = (q, r, d_1, d_2)$	0.58
$o'_{1,best} = (-4, 0, -6, -6)$	
$[T_2 \wedge D_2]_{o'}$	
$o' = (q, r, d_1, d_2)$	0.55
$o'_{2,best} = (-7, 0, 9, 13)$	
$Total : \vee_{i=1}^{2} [T_i \wedge D_i]_{o'_{i,best}}$	0.59

To illustrate the effectiveness of data generation from the mutated specifications, Table 3 displays the results of the conventional method that generates the test data directly from the original specifications. For the original specification, we generate test data only from the test condition T_i consisting of both pre-condition S_{pre} and guard-condition G_i meanwhile ignoring the defining condition D_i because the defining condition D_i involves unknown output variables that can not directly help to generate test data.

Table 3. Results for process Mod with original specifications.

Original Specification	Grade
$T_1 : x > 0 \wedge y \neq 0$	0.32
$T_2 : x < 0 \wedge y \neq 0$	0.38
$Total : \vee_{i=1}^{2} T_i$	0.37

For the proposed method, the final *Kill_rate* of $\vee_{i=1}^{2}[T_i \wedge D_i]_{o'_{i,best}}$ is 100%, the same as the conventional method. It means that every program mutant has been killed by at least one piece of test data. The corresponding final *Grade* is 0.59, larger than the *Grade* of 0.37 with the original specification-based method, indicating that the test data generated from $\vee_{i=1}^{2}[T_i \wedge D_i]_{o'_{i,best}}$ are of high quality that are more likely to kill all the program mutants. The result suggests that it is plausible to use these best individuals of chromosomes to make four mutated specifications for test case generation in the further maintenance of the original program.

Comparing the reformed specifications with the original ones in Figure 5, we can find the *Grade* of reformed ones that are always larger than that of original ones. It means that the data suite generated from the mutated specifications is more likely to pinpoint bugs than that of original ones, although both of them share the same *Kill_rate* of 100%.

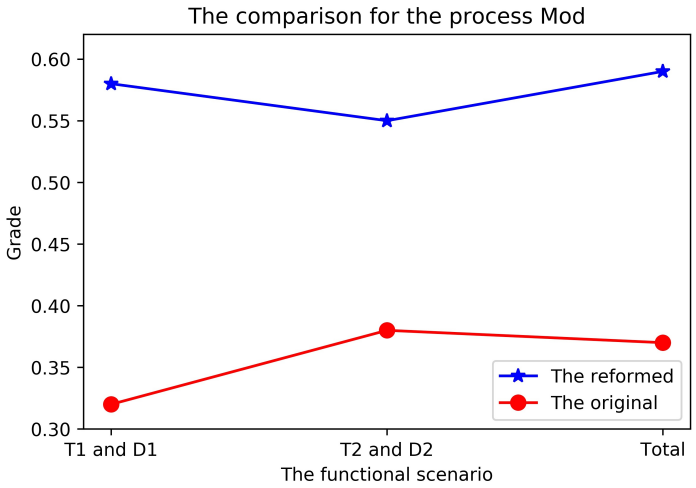

Figure 5. The grade of the mutated and original.

The Effect with Dummy Variables

We conduct additional experiments to figure out how dummy variables introduced into the different parts of defining conditions would affect the quality of the obtained mutated specifications. We make three versions of modifications to our approach as follows:

1. Version *V1*: Introducing dummy variables into only inequality relation;
2. Version *V2*: Introducing dummy variables into both equality and inequality relation;
3. Version *V3*: Putting no dummy variables in defining conditions.

For convenience, the approach with no modification, that is, with dummy variables for only equality relation, is called Version *V0*.

The previous experimental result for *V0* and the original, as well as the results from after applying variations of the approach *V1*,*V2*, and *V3* to process *Mod*, are together displayed in Figure 6.

According to Figure 6, three approaches with dummy variables *V0*, *V1*, and *V2* gain higher *Grade*s than the approach without dummy variables *V3*, and even *V3* seems to behave better than the conventional method. There are no significant differences between the evaluation of *V0* and *V2*. However, *V2* would occupy more computation resources than *V0* due to the consideration of more dummy variables. It seems that *V1* gains a little better final *Grade* than *V0*, though its *Grade* for each single mutated functional scenario is not good enough.

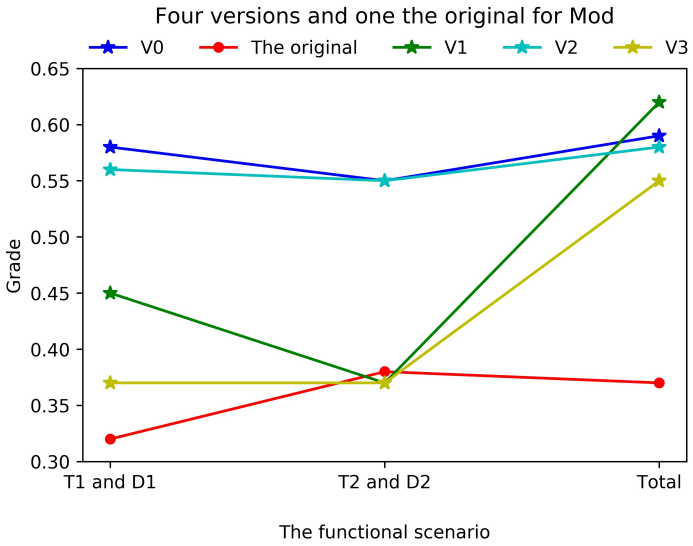

Figure 6. Results by four versions and the original for Mod.

In addition, by using an approach without dummy variables *V1* and *V3*, every obtained single mutated functional scenario demonstrates the strong capability to kill some specific program mutants while leaving other program mutants not killed, though the combination of all the functional scenarios in *V1* can reach 100% total *Kill_rate* while, for *V3*, the total *Kill_rate* unfortunately remains in 87.5%. This result demonstrates the importance of introducing dummy variables into equality relation in order to accomplish both good single and total *Grade*s and *Kill_rate*s.

In summary, it is necessary to introduce dummy variables into equality relations, and the additional dummy variables for inequality relation cannot significantly improve the proposed approach.

5.2. Case Study 2: Gcd

Process *gcd* is to compute the greatest common divisor of two inputs by using Stein's algorithm.

The formal specifications of *gcd* is:

process gcd (x: int, y: int) r: int
pre $x \geq 0 \wedge y \geq 0$
post $x > 0 \wedge y > 0 \wedge x \geq y \wedge r = gcd(y, x\%y) \vee$
$x > 0 \wedge y > 0 \wedge x < y \wedge r = gcd(y, y\%x) \vee$
$y = 0 \wedge r = x \vee$
$x = 0 \wedge r = y$
end_process

The implementation of process gcd in Python is:

```
def gcd(x, y):
if x < y:
x, y = y, x
if (0 == y):
return x
if x % 2 == 0 and y % 2 == 0:
return 2 * gcd(x//2, y//2)
if x % 2 == 0:
return gcd(x // 2, y)
if y % 2 == 0:
return gcd(x, y // 2)
return gcd((x - y) // 2, y)
```

Process *gcd* is a recursive process and its post-condition contains itself, so it is difficult to generate data from this kind of post-condition. We transform the original post-condition to the following ones:

$T_1 \wedge D_1 := x > 0 \wedge y > 0 \wedge x \geq y \wedge x\%r = 0 \wedge y\%r = 0 \wedge x\%y\%r = 0;$
$T_2 \wedge D_2 := x > 0 \wedge y > 0 \wedge x < y \wedge x\%r = 0 \wedge y\%r = 0 \wedge y\%x\%r = 0;$
$T_3 \wedge D_3 := x \geq 0 \wedge y = 0 \wedge r = x;$
$T_4 \wedge D_4 := y \geq 0 \wedge x = 0 \wedge r = y.$

Table 4 shows the chromosomes of process *gcd*.

Apply the algorithm to all of the chromosomes; in the meantime, make use of the original post-condition to determine whether the outputs of codes are correct or not. The results are displayed in Table 5.

The final *Kill_rate* of $\vee_{i=1}^{4}[T_i \wedge D_i]_{o'_{i,best}}$ is 100%. The corresponding *Grade* is 0.46, which means roughly 46 percent of test data that are randomly generated from $\vee_{i=1}^{4}[T_i \wedge D_i]_{o'_{i,best}}$ can kill all the program mutants.

Table 4. Chromosome forms for functional scenarios of process gcd.

Chromosome	D-Chromo	Dummy Vars
$[T_1 \wedge D_1]_{o'} : x \geq y \wedge$ $(d_1 \leq x\%r \leq d_2) \wedge$ $(d_3 \leq y\%r \leq d_4) \wedge$ $(d_5 \leq x\%y\%r \leq d_6)$	$o' =$ $(r, d_1, d_2, d_3$ $d_4, d_5, d_6)$	$d_1, d_2, d_3,$ d_4, d_5, d_6
$[T_2 \wedge D_2]_{o'} : x < y \wedge$ $(d_1 \leq x\%r \leq d_2) \wedge$ $(d_3 \leq y\%r \leq d_4) \wedge$ $(d_5 \leq y\%x\%r \leq d_6)$	$o' =$ $(r, d_1, d_2, d_3$ $d_4, d_5, d_6)$	$d_1, d_2, d_3,$ d_4, d_5, d_6
$[T_3 \wedge D_3]_{o'} : y = 0 \wedge$ $(d_1 \leq x - r \leq d_2)$	$o' =$ (r, d_1, d_2)	d_1, d_2
$[T_4 \wedge D_4]_{o'} : x = 0 \wedge$ $(d_1 \leq y - r \leq d_2)$	$o' =$ (r, d_1, d_2)	d_1, d_2

Table 5. Results for process gcd after applying GA.

The Best Individual of Chromosome	Grade
$[T_1 \wedge D_1]_{o'}$ $o'_{1,best} =$ $(0, 1, 10, 10, 10, 3, 6)$	0.72
$[T_2 \wedge D_2]_{o'}$ $o'_{2,best} =$ $(8, 4, 6, 2, 2, 0, 8)$	0.58
$[T_3 \wedge D_3]_{o'}$ $o'_{3,best} = (5, -1, 20)$	0.0037
$[T_4 \wedge D_4]_{o'}$ $o'_{4,best} = (4, -3, 16)$	0.0037
$Total : \vee_{i=1}^{4}[T_i \wedge D_i]_{o'_{i,best}}$	0.46

Conversely, the result for applying the method that generates test data directly from the original specification is displayed in Table 6.

Table 6. Results for process gcd with original specifications.

Original Specification	Grade
$T_1 : x > 0 \wedge y > 0 \wedge x \geq y$	0.29
$T_2 : x > 0 \wedge y > 0 \wedge x < y$	0.49
$T_3 : x \geq 0 \wedge y = 0$	0.0035
$T_4 : y \geq 0 \wedge x = 0$	0.0035
$Total : \vee_{i=1}^{2} T_i$	0.25

Comparing the reformed specifications with the original ones in Figure 7, we can find that the first two reformed ones $[T_1 \wedge D_1]_{o'}$ and $[T_2 \wedge D_2]_{o'}$ have very high values of *Grade*, 0.72 and 0.58, respectively, higher than 0.29 and 0.49 with the original specifications. In addition, the *Kill_rate* of a sole $[T_i \wedge D_i]_{o'}$ $(i = 1, 2)$ is 94%, indicating that the test data generated from the first two reformed specifications are likely to pinpoint most bugs probably occurring in the program. Only a few program mutants (6% of total), with some faults that directly relates to the last two functional scenarios $T_3 \wedge D_3$ and $T_4 \wedge D_4$ (where $x = 0$ or $y = 0$), cannot be killed by the test data generated from either the first two reformed specifications or the first two original specifications. Due to the very simple forms and the limited functionality of the last two functional scenarios, there is no improvement of test data generation using our method against the original ones.

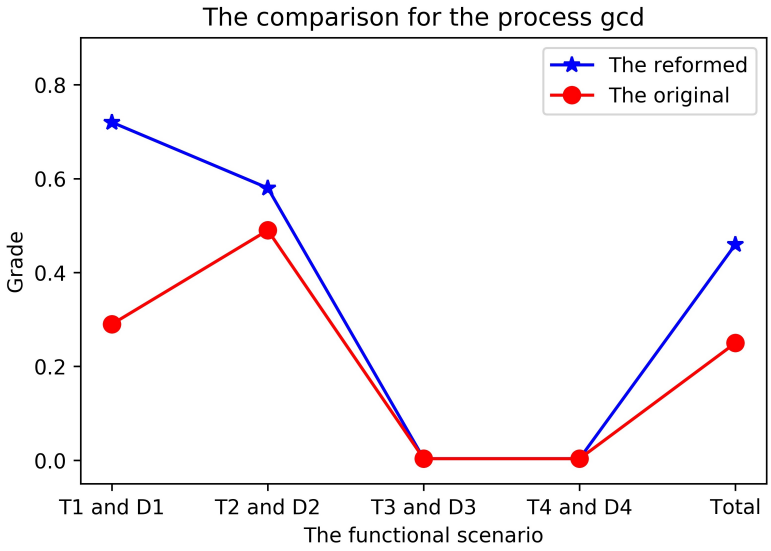

Figure 7. The grade of the reformed and original.

The results from both classic examples demonstrate that the input data generated from the mutated specifications are more likely to kill the mutants of programs than that from the original specifications.

The Effect without Dummy Variables

Like what we have done for process *Mod*, we conduct additional experiments with the approach without using dummy variable V3 since *gcd* only has equality relations in the defining conditions. The results are shown in Figure 8.

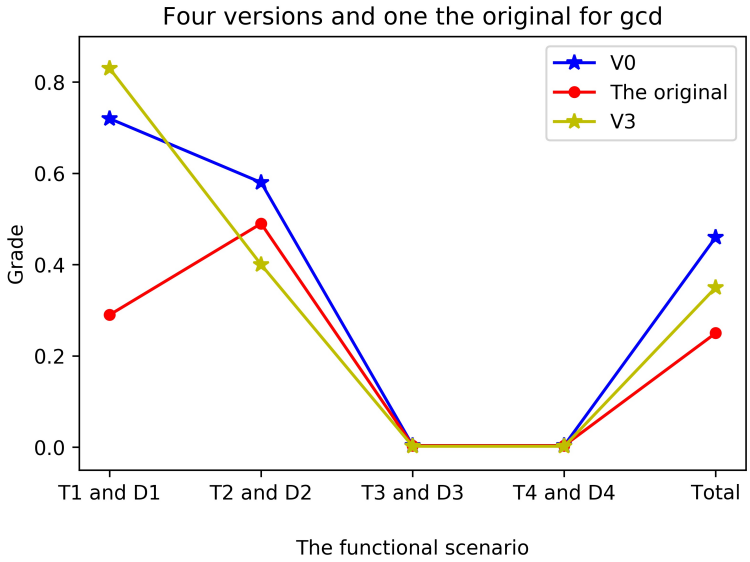

Figure 8. Results by four versions and the original for gcd.

The experimental results are similar to that in *Mod*. $V0$ performs better than $V3$. $V3$ still encounters the problem that every single mutated functional scenario is not able to kill all the program mutants. It shows that the test data generated from those strict equality relations are less likely to trigger some bad behaviors of program.

5.3. Complexity of Our Approach

We present an abstract analysis of the complexity for our approach. Generally, a GA complexity is on the order of $O(g * n * m)$ without the effect of the fitness function, where g is the number of generations, n is the population size, and m is the number of functional scenarios. Since our approach uses a fitness function involved in the mutation testing, we should take both the program execution time and the data suite generation time into consideration.

As the speed of the constraint solver to solve an individual formula (to generate a test suite) depends on the complexity of the functional scenarios (logical formulas) whose complexity cannot be easily determined, we associate the cost of using the constraint solver for a singular individual with the number of input variables in, the number of output variables out, and the number of dummy variables d. In addition, the number of dummy variables relies on the number of equality relation in each functional scenario, which varies in different kinds of programs. We simply assume that each functional scenario has at least one equality relation. Thus, the complexity of using the constraint solver for each individual is $O(in + out + 2 * d * m)$. Moreover, the complexity of all the executions for program mutants is approximately $O(mu * sui)$ with mu the number of program mutants and sui the size of test data suite. Finally, considering the complexity of the GA together with the mutation testing, the complexity for our approach is

$$O(g * n * ((in + out + 2 * d * m) * (mu * sui)) * m).$$

6. Conclusions

We propose a new method for effective test data generation based on mutated pre-post style formal specifications. The method is characterized by the integration of the functional scenario-based testing, a genetic algorithm and the mutation testing. In the approach, by assigning appropriate values to the unknown output and dummy variables to the variations for the original specifications, we can obtain useful mutated specifications that are sensitive to small syntactic structural changes of program codes.

We have also carried out two classic cases to evaluate the performance of our method. The results of case studies demonstrate that, for a complicated functional scenario, the proposed approach is capable of effectively generating useful test data to kill as many program mutants as possible, which outperforms the conventional data generation method.

In spite of the advantages of our method as mentioned above, there are also some limitations and disadvantages in the application of our method. First, the proposed method can only work on arithmetical relationships between inputs and outputs in which outputs affect the generation of inputs. Second, as the GA usually iterates many times and executes all the program mutants for every iteration, the cost would not be low. However, if we have enough computing resources for applying our method, it might be worth taking time to obtain good reformed specifications for the further maintenance of software.

In order to cope well with complex real programs, some additional extensions can be made in our approach. Firstly, by using the character encoding standard like US-ASCII [45], we can convert a String to a byte array so that the relationship that contains string variables can also be manipulated by our method. Moreover, since many research works exist concerning about the techniques of encoding complex data [46–48] that may occur in specifications like images and videos, it is possible to transform these specifications into appropriate arithmetical relationships so that our approach can be used in such cases. Secondly, although there exist specifications where the input and output variables are not specified by some explicit arithmetical equality relation, our method would still be applicable. Because instead of directly using these specifications, we can design some mutated arithmetical relationships (in form of inequality) of input and output that can not only approximate to the real properties of program but also leave open the possibility of occurrence of unexpected behaviors. Thirdly, when testing a big complex system, we can decompose it into a set of subroutines and focus on testing small procedures one by one using our approach. Thus, there is no need to repeatedly executing the whole system with our algorithm.

In future work, we will focus on enhancing the capability of this method to deal with more kinds of relationships between inputs and outputs where the values of outputs may not directly determine the inputs. We will conduct more experiments to ensure that our method can be well used in different kinds of programs.

Author Contributions: Conceptualization, R.W., S.L., and Y.S.; methodology, R.W. and Y.S.; investigation, R.W.; resources, R.W.; data curation, R.W.; writing—original draft preparation, R.W.; writing—review and editing, S.L. and Y.S.; visualization, R.W.; supervision, S.L. and Y.S.; project administration, S.L. and Y.S.; funding acquisition, S.L. and Y.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by JSPS KAKENHI Grant No. 26240008.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

In Algorithm A1, the function *one_step* creates a new population with fitness values from the previous population through applying crossover and mutation operations; and the function *do_valuation* assigns fitness values calculated by the function *Grade* to all of the individuals by using the feedback of testing program mutants.

Algorithm A1 GA to obtain mutated specifications.

Inputs: the functional scenarios from the specification: $T_i \wedge D_i$
Individuals: $o^{'} = (o_{1,} \cdots, o_n, d_{1,\ldots,} d_m)$ with concrete values
Outputs: the reformed specification $[T_i \wedge D_i]_{o'}$

```
run(){
    result = list()
    for [T_i ∧ D_i]_o' in functional scenarios:
        spec = T_i ∧ D_i
        population = initial(o')
        while(not enough iterations){
            one_step(spec)
      }
        best_individual =
                select_best_individual(population)
        reformed_specification = (spec, best_individual)
        result.append(reformed_specification)
}
one_step(spec) {
    # This function selects top 50% of the current population
    population = keep_good_individuals(population)
    do:
        father, mother = random_select_two(population)
        child1, child2 = crossover_operation(father,mother)
        child1, child2 = mutation_operation(child1,child2)
        population.put(child1,child2)
    until population increases enough
    do_valuation(population,spec)
}
do_valuation(population,spec){
    for individual in population:
        datas = data_suite_from(individual,spec)
        statistic_sum = kill_program_mutants(datas)
        individual.value = Grade(statistic_sum)
}
```

References

1. Wong, W.E.; Horgan, J.R.; London, S.; Agrawal, H. A study of effective regression testing in practice. In Proceedings of the Eighth International Symposium on Software Reliability Engineering, Albuquerque, NM, USA, 2–5 November 1997; pp. 264–274.
2. Leung, H.K.; White, L. Insights into regression testing (software testing). In Proceedings of the Conference on Software Maintenance, Miami, FL, USA, 16–19 October 1989; pp. 60–69.
3. Kazmi, R.; Jawawi, D.N.; Mohamad, R.; Ghani, I. Effective regression test case selection: A systematic literature review. *ACM Comput. Surv. (CSUR)* **2017**, *50*, 1–32. [CrossRef]
4. Stocks, P.; Carrington, D. A framework for specification-based testing. *IEEE Trans. Softw. Eng.* **1996**, 777–793. [CrossRef]
5. Richardson, D.; O'Malley, O.; Tittle, C. *Approaches to Specification-Based Testing*; ACM: New York, NY, USA, 1989; Volume 14.
6. Khurshid, S.; Marinov, D. TestEra: Specification-based testing of Java programs using SAT. *Autom. Softw. Eng.* **2004**, *11*, 403–434. [CrossRef]
7. Hierons, R.M.; Bogdanov, K.; Bowen, J.P.; Cleaveland, R.; Derrick, J.; Dick, J.; Gheorghe, M.; Harman, M.; Kapoor, K.; Krause, P.; et al. Using formal specifications to support testing. *ACM Comput. Surv. (CSUR)* **2009**, *41*, 1–76. [CrossRef]
8. Dokhanchi, A.; Hoxha, B.; Fainekos, G. Formal requirement debugging for testing and verification of cyber-physical systems. *ACM Trans. Embed. Comput. Syst. (TECS)* **2017**, *17*, 1–26. [CrossRef]
9. Offutt, A.J.; Liu, S. Generating Test Data from SOFL Specifications. *J. Syst. Softw.* **1999**, *49*, 49–62. [CrossRef]
10. Dick, J.; Faivre, A. Automating the generation and sequencing of test cases from model-based specifications. In Proceedings of the International Symposium of Formal Methods Europe, Odense, Denmark, 19–23 April 1993; Springer: Berlin/Heidelberg, Germany, 1993; pp. 268–284.

11. Ed-Douibi, H.; Izquierdo, J.L.C.; Cabot, J. Automatic generation of test cases for REST APIs: A specification-based approach. In Proceedings of the 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC), IEEE, Stockholm, Sweden, 16–19 October 2018; pp. 181–190.
12. Alrawashed, T.A.; Almomani, A.; Althunibat, A.; Tamimi, A. An Automated Approach to Generate Test Cases From Use Case Description Model. *Comput. Model. Eng. Sci.* **2019**, *119*, 409–425. [CrossRef]
13. Jones, C.B. *Systematic Software Development Using VDM*; Citeseer: Princeton, NJ, USA, 1990; Volume 2.
14. Larsen, P.G.; Battle, N.; Ferreira, M.; Fitzgerald, J.; Lausdahl, K.; Verhoef, M. The overture initiative integrating tools for VDM. *ACM SIGSOFT Softw. Eng. Notes* **2010**, *35*, 1–6. [CrossRef]
15. Tran-Jørgensen, P.W.; Nilsson, R.S.; Lausdahl, K. Enhancing Testing of VDM-SL models. In Proceedings of the 16th Overture Workshop, Oxford, UK, 14 July 2018; pp. 7–22.
16. Liu, S. *Formal Engineering for Industrial Software Development: Using the SOFL Method*; Springer Science & Business Media: Berlin, Germany, 2013.
17. Liu, S.; Nakajima, S. 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), LNCS 8332, Queenstown, New Zealand, 29 October 2013; Springer: Queenstown, New Zealand, 2013; pp. 3–16.
18. Wang, R.; Sato, Y.; Liu, S. Specification-based Test Case Generation with Genetic Algorithm. In Proceedings of the 2019 IEEE Congress on Evolutionary Computation (CEC), Wellington, New Zealand, 10–13 June 2019; pp. 1382–1389.
19. Rapps, S.; Weyuker, E.J. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.* **1985**, 367–375. [CrossRef]
20. Weyuker, E.J. More experience with data flow testing. *IEEE Trans. Softw. Eng.* **1993**, *19*, 912–919. [CrossRef]
21. Khedker, U.; Sanyal, A.; Sathe, B. *Data Flow Analysis: Theory and Practice*; CRC Press: Boca Raton, FL, USA, 2017.
22. Pargas, R.P.; Harrold, M.J.; Peck, R.R. Test-data generation using genetic algorithms. *Softw. Test. Verif. Reliab.* **1999**, *9*, 263–282. [CrossRef]
23. Girgis, M.R. Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm. *J. UCS* **2005**, *11*, 898–915.
24. Girgis, M.R.; Ghiduk, A.S.; Abd-Elkawy, E.H. Automatic generation of data flow test paths using a genetic algorithm. *Int. J. Comput. Appl.* **2014**, *89*, 29–36.
25. Nayak, N.; Mohapatra, D.P. Automatic test data generation for data flow testing using particle swarm optimization. In Proceedings of the International Conference on Contemporary Computing, Noida, India, 9–11 August 2010; Springer: Cham, Switzerland, 2010; pp. 1–12.
26. Biswas, S.; Kaiser, M.S.; Mamun, S. Applying ant colony optimization in software testing to generate prioritized optimal path and test data. In Proceedings of the 2015 International Conference on Electrical Engineering and Information Communication Technology (ICEEICT), IEEE, Dhaka, Bangladesh, 21–23 May 2015; pp. 1–6.
27. Harman, M.; Jia, Y.; Langdon, W.B. 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, ACM, Szeged, Hungary, 5–9 September 2011; pp. 212–222.
28. Papadakis, M.; Kintis, M.; Zhang, J.; Jia, Y.; Le Traon, Y.; Harman, M. Mutation testing advances: An analysis and survey. In *Advances in Computers*; Elsevier: Amsterdam, The Netherlands, 2019; Volume 112, pp. 275–378.
29. Ma, Y.S.; Offutt, J.; Kwon, Y.R. MuJava: An automated class mutation system. *Softw. Test. Verif. Reliab.* **2005**, *15*, 97–133. [CrossRef]
30. Derezinska, A.; Kowalski, K. Object-oriented mutation applied in common intermediate language programs originated from c. In Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, Germany, 21–25 March 2011; pp. 342–350.
31. Delgado-Pérez, P.; Medina-Bulo, I.; Palomo-Lozano, F.; García-Domínguez, A.; Domínguez-Jiménez, J.J. Assessment of class mutation operators for C++ with the MuCPP mutation system. *Inf. Softw. Technol.* **2017**, *81*, 169–184. [CrossRef]
32. Jackson, D. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2002**, *11*, 256–290. [CrossRef]
33. Sullivan, A.; Wang, K.; Zaeem, R.N.; Khurshid, S. Automated test generation and mutation testing for Alloy. In Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), Tokyo, Japan, 13–17 March 2017; pp. 264–275.
34. Martins, E.; Sabião, S.B.; Ambrosio, A.M. ConData: A tool for automating specification-based test case generation for communication systems. *Softw. Qual. J.* **1999**, *8*, 303–320. [CrossRef]
35. McMillan, K.L.; Zuck, L.D. Formal specification and testing of QUIC. In *Proceedings of the ACM Special Interest Group on Data Communication*; ACM: New York, NY, USA, 2019; pp. 227–240.
36. Ali, S.; Iqbal, M.Z.; Arcuri, A.; Briand, L.C. Generating test data from OCL constraints with search techniques. *IEEE Trans. Softw. Eng.* **2013**, *39*, 1376–1402. [CrossRef]
37. Jalila, A.; Mala, D.J. Automated optimal test data generation for OCL specification using harmony search algorithm. *Int. J. Bus. Intell. Data Min.* **2020**, *16*, 231–259. [CrossRef]
38. Holland, J.H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*; MIT Press: Cambridge, MA, USA, 1992.

39. DeMillo, R.A.; Lipton, R.J.; Sayward, F.G. Hints on test data selection: Help for the practicing programmer. *Computer* **1978**, *11*, 34–41. [CrossRef]
40. Luo, J.; Liu, S.; Wang, Y.; Zhou, T. Applying SOFL to a railway interlocking system in industry. In Proceedings of the International Workshop on Structured Object-Oriented Formal Language and Method, Tokyo, Japan, 15 November 2016; Springer: Cham, Switzerland, 2016; pp. 160–177.
41. Liu, S. *Formal Engineering for Industrial Software Development Using the SOFL Method*; Springer: Berlin, Germany, 2004; ISBN 3-540-20602-7.
42. Sen, K. Concolic testing. In Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ACM, Atlanta, GA, USA, 5–9 November 2007; pp. 571–572.
43. Sato, Y.; Sugihara, T. Automatic Generation of Specification-Based Test Cases by Applying Genetic Algorithms in Reinforcement Learning. In Proceedings of the International Workshop on Structured Object-Oriented Formal Language and Method, Paris, France, 6 November 2015; Springer: Cham, Switzerland, 2015; pp. 59–71.
44. De Moura, L.; Bjørner, N. Z3: An efficient SMT solver. In Proceedings of the International conference on Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary, 29 March–6 April 2008; Springer: Cham, Switzerland, 2008; pp. 337–340.
45. Mackenzie, C.E. *Coded-Character Sets: History and Development*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1980.
46. Basavaprasad, B.; Ravi, M. A study on the importance of image processing and its applications. *IJRET Int. J. Res. Eng. Technol.* **2014**, *3*, 1.
47. Barannik, V.; Podlesny, S.; Tarasenko, D.; Barannik, D.; Kulitsa, O. The video stream encoding method in infocommunication systems. In Proceedings of the 14th International Conference on Advanced Trends in Radioelecrtronics, Telecommunications and Computer Engineering (TCSET), IEEE, Lviv-Slavske, Ukraine, 20–24 February 2018; pp. 538–541.
48. Hur, T.; Bang, J.; Huynh-The, T.; Lee, J.; Kim, J.I.; Lee, S. Iss2Image: A novel signal-encoding technique for CNN-based human activity recognition. *Sensors* **2018**, *18*, 3910. [CrossRef] [PubMed]