Specification-based Test Case Generation with Genetic Algorithm

Rong Wang
Department of Computer Science
Hosei University
Tokyo, Japan
rong.wang.99@stu.hosei.ac.jp

Yuji Sato
Department of Computer Science
Hosei University
Tokyo, Japan
yuji@hosei.ac.jp

Shaoying Liu
Department of Computer Science
Hosei University
Tokyo, Japan
sliu@hosei.ac.jp

***Abstract*—As the current specification-based testing (SBT) face some challenges in test case generation for regression testing, we propose a new method for efficient test case generation that combines formal specifications with genetic algorithm (GA). This method mainly reforms formal specifications though GA to generate inputs that can kill as many as possible mutants of the target program under test. For case study, two classic examples are presented to demonstrate the efficiency of this method. The result shows that this method is able to help efficiently generate useful test cases to uncover all the program mutants, which contributes to further maintenance of software.**

***Index Terms*—test data generation, genetic algorithm, specification-based testing, regression testing, mutant testing**

I. Introduction

Regression testing is an important technique to ensure that previously tested software still performs the same way after it is changed or interfaced with other software [1][2]. 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) can be effectively used for regression testing.

Specification-based testing is characterized by test data being generated from the specification without considering the structure of the corresponding program and test result being analyzed based on the specification [3] [4] [5]. Formal specifications may allow the test data generation and test result analysis to be done rigorously, systematically, and even automatically in many cases [6] [7]. However, in spite of considerable progresses having been made, SBT still has deficiencies. One of the major deficiencies is that test data generation only considers constraints on inputs of operations, which does not take advantage of constraints on outputs.

Let us focus our discussion on formal specifications in pre- and post-conditions, such as Vienna Development Method (VDM) [8] and Structured-Object-based-formal Language (SOFL) [9]. The most advanced technique for test data generation from formal specifications is known as *functional scenario-based test data generation* [10]. 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*. The test condition only involves input variables of the operation while the defining condition must involve some output variables. 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 and leaves the defining condition untouched [6] [11] [12]. 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.

To overcome the limitations of the current methods, we propose a new method for test data generation in this paper. This method is to obtain the enhanced formal specifications by using GA for generating input data that are more likely to kill mutants of the target program under test. The new method features the combination of SBT, mutant testing, and genetic algorithm (GA) in the way that GA is used to find the best mutant test condition from the specification and SBT is used to generate test data from the mutant test condition. In this method, 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. The expected effect of the test data generated in this way is to find more bugs in the code.

We use SOFL as the formal notation for specifications in this paper for two reasons. One is that 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 that SOFL is familiar to us and its use in industry has been increasing [13].

The method has two key characteristics.

First, compared to the traditional specification-based test data generation, our method not only considers both pre-conditions and guard-conditions for test case generation as

978-1-7281-2153-6/19/$31.00 ©2019 IEEE

traditional methods do, but also concerns the defining conditions that contains the relationships of inputs and outputs. Second, we introduce dummy variables into defining conditions so as to make defining conditions more flexible for test case generation meanwhile keeping the similar properties of original specifications.

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

II. Related Work

Currently, there are some different techniques widely used in testing unrelated of formal specifications.

The technique proposed in [14] is used to automatically search for test data by using genetic algorithm with control-dependence graph 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. Compared with this technique, the genetic algorithm used in our method is to produce good reforming specifications for test case generation, while the genetic algorithm in [5] is to directly search for good test data.

Paper [15] introduced a mutation-based test data generation approach, which targets strong mutation adequacy and is capable of killing both first and higher order mutants. In our method, we also use program mutants to guide the evolution of the genetic algorithm.

Equivalence partitioning (EP) is one kind of black-box testing, which divides the input domain of a program onto equivalence classes [16]. In EP, for a given test coverage criterion, test case sets take the form of simultaneous equalities and inequalities on the module inputs.

Similar to EP, another type of the testing, called Boundary Value Analysis (BVA), is mainly focused on the input domain. There are two types of boundary value analysis, one is called normal boundary value testing that concerns about only valid variable values, the another is called robust boundary value testing that considers both valid and invalid variable values.

Different from our method, both BVA and EP usually consider only input domain without making use of the information about the relation between inputs and outputs, while our method consider the defining condition as a good hint for generating inputs.

Data flow testing techniques is a type of white-box testing, which considers how program variables are defined and used [17]. All of these techniques use the data flow information in a program to guide the selection of test data. Compared to our method, data flow testing techniques look very deep into the program code, while our method focuses on the specifications without analyzing the code itself.

There are also some techniques related to formal specification-based testing (SBT).

A thesis [18] provides a systematic review on researches concerning Automated Test Data Generation (ATDG) techniques in the period 1997-2006. Moreover it also aims at identifying probable gaps in research about ATDG techniques of defined period so as to suggest the scope for further research.

Offutt and Liu [6] describe 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.

TestEra [5] is 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 post-conditions of that method, and a bound that limits the size of the test cases to be generated. Using the method's pre-condition, TestEra automatically generates all nonisomorphic test inputs up to the given bound. It executes the method on each test input, and uses the method postcondition as an oracle to check the correctness of each output.

ConData [19] describes a tool called ConData used as test generation for communication protocols specified as extended finite state machines. The strategy for test generation combines different specification-based test methods: (i) transition testing for the control part of a protocol and (ii) syntax and equivalence partitioning for the data part. The tool uses a representation of the protocol in PSL (Protocol Specification Language), which is transformed into a format readable by a Prolog program.

Paper [20] describe an approach to testing programs based on data-flow-oriented specifications by analyzing data flow paths and discussing criteria for test case generation. This approach suggests a specific way to generate test cases directly from formalized data flow diagrams and the associated textual specifications.

Paper [21] define a set of mutation operators and implement a mutation generator for specifications written in SMV, a popular model checker. This paper mainly focus on state-based specifications.

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, only using the relationship of inputs from specifications could not be sufficient to find out the bugs. Different from these techniques, we make use of relationship among inputs and outputs and focus on automatically reforming the formal specifications SOFL that has pre-post style by using GA, so that the inputs generated from these advanced specifications are good enough to be more likely to kill mutants of the program.

III. Reforming specifications with GA algorithm

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

A. GA algorithm

A genetic algorithm (GA) is a heuristic search method inspired from evolutionary biology and is first proposed by John Holland [22]. A GA has three key procedures : i) it creates a population of individuals represented by a pre-defined chromosome that are typically encoded the solutions to a problem; ii) it selects a group of optimal individuals for next generation by an undetermined selection strategy; iii) it repeats the first two steps until the remaining individuals are good enough according to both evaluation function and stop criteria.

During the evolution of the population, there are also three primary concepts of genetic manipulation for individuals, selection, crossover and mutation respectively. Later we would carefully describe these three operators used in our method.

Since GA is good at finding optimal solutions for non-linear problems and the specifications of pre-post style could be easily transformed to chromosomes with few efforts, we apply GA for finding the best reformed specifications in this paper. Later we would describe how to transform the specifications into a chromosome, then give the details of genetic manipulations that are used in our method.

B. Mutant Testing

Mutant testing, also called program mutation [23], 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 behaviours of program mutants different from that of the original program.

In our approach, Mutant testing is also involved in the evaluation of generated test cases, for obtaining more flexible and useful reformed specifications.

C. Reforming Specifications

In pre-post style formal specifications, the *defining condition* that describes the constraints of inputs and outputs after a method in the program performs, is not used to guide the generation of input data. However, as the defining condition indicates the behaviours of a program, it is necessary to make use of defining condition to generate test data keen to the behaviours of the program.

Generally, the defining condition is not used for directly generating input data because the values of outputs in defining condition are unknown to us.

Since defining conditions describe how output variables relate to input variables, they are often used to check whether a run of the program is correct or not, rather than used to directly generate the inputs. For a complex program, it is difficult to directly generate inputs to satisfy a defining condition without knowing their 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 ($x*y > x+y$) due to the unknown output y, So 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 $x*2 > x+2$ (when $y = 2$) may be more likely to trigger bugs than that of $x * 1 > x + 1$ (when $y = 1$). We are not sure about which is better. Our work is mainly concentrate on deciding the output values for various programs. 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 such good values for outputs from the defining condition.

Moreover, to enhance the capability of the reformed specifications mentioned above, some extension would also be added into defining conditions. In this extension, the defining condition is slightly changed to be able to consider more bad behaviours that are possibly occurred in the program.

To sum, in our work, *reforming specifications* means changing specifications according to certain rules. More precisely, the reformed specifications are obtained by two rules:

1. Assigning useful concrete values to output variables in the defining conditions;
2. Slightly change the form of defining conditions to allow bad behaviours to be occurred.

Our goal of reforming a specification is to obtain a new version of the specification from which test data suit useful for uncovering bugs in the program can be easily generated. In this section, we first define the chromosome forms for the formal specifications of a process, as well as the crossover operation and mutation operation for chromosomes, then we apply the genetic algorithm to obtain suitable reformed specifications for test data generation.

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 [24]. 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 eliminated in real CDFDs. Therefore, our discussion on test data generation from specifications focuses on a single process.

D. Chromosome

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* S_{pre} *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.

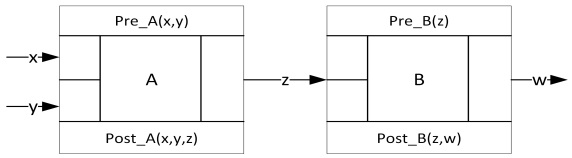

Fig. 1. Process A and Process B

Now we explain how to make slight extension to change the form of defining conditions so as to allow bad behaviours to be occurred. To obtain a more flexible and useful reformed specifications, we introduce *dummy variables, say* $d_i(i = 1, \cdots, c)$, to equations that relate inputs and outputs in a defining condition. Then we combine both *dummy variables* and output variables as an output vector that are needed to be assign concrete values.

Definition 2 *Output vector $o^{'}$. 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 variables.*

In our work, that is, for an equation $f(inputs, outputs) = 0$ from any defining condition D_i, modify it into an inequality

$d_1 <= f(inputs, outputs) <= d_2$

Then the output vector $o^{'} = (o_{1,} \cdots, o_n, d_{1,} d_2)$.

Definition 3 *A chromosome* $[T_i \wedge D_i]_{o'}\ (i = 1, \cdots, N)$ *is a reformed functional scenario* $T_i \wedge D_i$ *based on two rules. An individual (a solution to the problem) represented by chromosome* $[T_i \wedge D_i]_{o'}$ *contains only symbolic inputs and concrete values for its output vector* $o^{'} = (o_{1,} \cdots, o_n, d_{1,} \cdots, d_c)$*. And a population is a group of individuals. Besides, output vector* $o^{'}$ *is also called d-chromo, each element of* $o^{'}$ *is called a genetic.*

From this definition, given a d-chromo $o^{'}$, an individual $[T_i \wedge D_i]_{o'}$ can be directly used for test data generation. It is clear that an individual $[T_i \wedge D_i]_{o'}$ is determined by $o^{'}$. We do genetic manipulation for d-chromo $o^{'}$ of an individual $[T_i \wedge D_i]_{o'}$ to produce new individuals, while we evaluate an individual $[T_i \wedge D_i]_{o'}$ (represented by a chromosome) by evaluating the test data generated from this individual.

E. Genetic manipulation

Genetic manipulation refers to the change of genetic structure in biology, but in genetic algorithm, it means a "child" solution is produced from a pair of "parent" solutions by using genetic operators such as crossover and mutation operations.

For crossover operation, a pair of individuals (solutions) from the current population are selected as parents to produce their offsprings. In details, first randomly 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 $p\ (0 < p < 1)$ and obtain two new individuals.

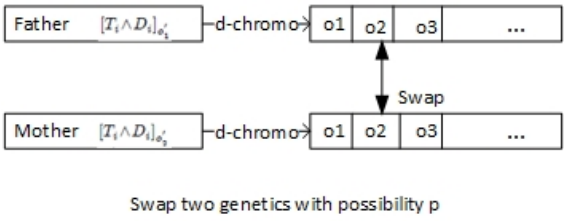

Fig. 2. Crossover operation

For mutation operation, each genetic of an individual is mutated with possibility $q\ (0 < q < 1)$. 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.

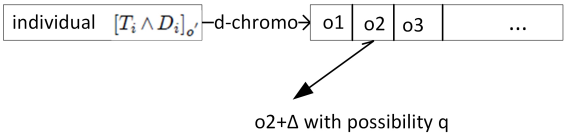

Fig. 3. Mutation operation

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 $N\_kill_{i,o'} = (k_1, ..., k_m)$ 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:

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

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

For a given chromosome $[T_i \wedge D_i]_{o'}$, its individual of appropriate de-chromo $o'_{i,best}$ is regraded the best if and only if this individual possesses the highest value of $Grade$ among the whole population. The GA is to find out the best individuals of d-chromo $o'_{i,best}$ for each chromosome $[T_i \wedge D_i]_{o'}$ $(i = 1, \cdots, N)$.

After all the individuals from the current population are evaluated, GA would select the most 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 the first 50 percent of individuals to form a new population.

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

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

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

The final formula is for the comparision between our approach and different other techniques of test data generation. In our case study, our method is compared with the conventional method in respect of efficiency of test data generation.

IV. Genetic Algorithm for reforming specifications

As mentioned previously, reforming specifications aims to produce reformed specifications for test data generation, but how to carry out the reforming so that reformed specifications of good quality can be produced is still a remaining issue to be addressed. In our work, we adopt a genetic algorithm for this purpose. Specifically, it takes the following steps:

1. Inject faults into the original program to form a set of program mutants.

2. Use input-related constraint conditions $[T_i \wedge D_i]_{o'}$ as seed chromosomes for the first generation of the population.

3. Apply the genetic algorithm to each chromosome and select the best individuals(the best solutions). Determine whether a test case from a mutant test condition kill the code mutants or not based on the original specifications. $\vee_{i=1}^{n}(T_i \wedge D_i)$.

The procedure of our approach with GA is displayed in Figure 4. Correspondingly, we provide the pseudo-code of the proposed approach in *Algorithm 1*.

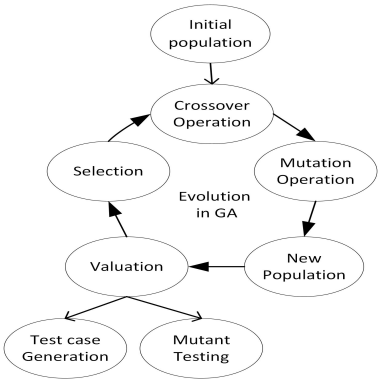

Fig. 4. The workflow of GA with specifications

In *Algorithm 1*, function one_step creates a new population with fitness values from the previous population through applying crossover and mutation operations; function $do\_valuation$ assigns fitness values to all of the individuals by using the feedback of testing program mutants; function $Grade$ is a valuation function that defined in the former section.

V. 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. In addition, we would also compare our method with the conventional method that directly generate input data from original specifications. The results demonstrate that the inputs generated from the reformed specifications do kill more mutants of the programs than that of original specifications.

A. 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*, both the formal specification in SOFL [24] and implementation in Java are given respectively as follows:

Formal specification of process Mod:

Case study 1.

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 .

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. Because

Algorithm 1 Genetic algorithm for reforming specification

$Inputs: spec \quad from \quad function \quad scenarios: T_i \wedge D_i$

$Individual: o^{'} = (o_1, \cdots, o_n, d_{1,\cdots,}d_m) with \quad concrete \quad values$

$Outputs: new \quad [T_i \wedge D_i]_{o'}$

```
run(){
    result = list()
    for [T_i ∧ D_i]_{o'} in function scenarios:
        spec = T_i ∧ D_i
        population = initial(o')
        while(not enough steps){
            one_step(spec)
        }
        best_individual =
                select_best_individual(population)
        reformed_specification = (spec, best_individual)
        result.append(reformed_specification)
}
one_step(spec) {
    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 increase enough
    do_valuation(population,spec)
}
do_valuation(population,spec){
    for individual in population:
        datas = generated_data(individual,spec)
        statistic_sum = kill_program_mutants(datas)
        individual.value = Grade(statistic_sum)
}
```

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 $-7 \ mod \ 5 = 3$.

First of all, obtain function scenarios $T_i \wedge D_i$ from the specification:

$T_1 \wedge D_1 := x > 0 \wedge y \neq 0 \wedge y = q * x + r \wedge Abs(r) < x \wedge xr \geq 0$

$T_2 \wedge D_2 := x < 0 \wedge y \neq 0 \wedge y = q * x + r \wedge$
$Abs(r) < -x \wedge xr \geq 0$

$T_3 \wedge D_3 := x \neq 0 \wedge y = 0 \wedge q = 0 \wedge r = 0$

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

Apply *Algorithm 1* to these chromosomes. The results are displayed in Table II.

TABLE I
CHROMOSOME FORMS FOR FUNCTION SCENARIOS OF PROCESS MOD

chromosome	d-chromo	dummy vars
$[T_1 \wedge D_1]_{o'} : x > 0 \wedge y \neq 0 \wedge$ $d_1 \leq q * x + r - y \leq \wedge d_2 \wedge$ $Abs(r) < x \wedge xr \geq 0$	$o^{'} =$ (q, r, d_1, d_2)	d_1, d_2
$[T_2 \wedge D_2]_{o'} : x < 0 \wedge y \neq 0 \wedge$ $d_1 \leq q * x + r - y \leq \wedge d_2 \wedge$ $Abs(r) < -x \wedge xr \geq 0$	$o^{'} =$ (q, r, d_1, d_2)	d_1, d_2

TABLE II
RESULTS FOR PROCESS MOD AFTER APPLYING GA

best chromosome	Grade	Kill_rate
$[T_1 \wedge D_1]_{o'}$ $o^{'} = (q, r, d_1, d_2)$ $o^{'}_{1,best} = (-9, 0, -13, 0)$	0.58	100%
$[T_2 \wedge D_2]_{o'}$ $o^{'} = (q, r, d_1, d_2)$ $o^{'}_{2,best} = (-2, 1, -10, 7)$	0.59	100%
$\vee_{i=1}^{2}[T_i \wedge D_i]_{o'_{i,best}}$	0.62	100%

The final $Kill\_rate$ of $\vee_{i=1}^{2}[T_i \wedge D_i]_{o'_{i,best}}$ representing the percentage of killed program mutants is 100%. And the corresponding $Grade$, is 0.62, means nearly 62 percent of test cases generated from $\vee_{i=1}^{2}[T_i \wedge D_i]_{o'_{i,best}}$ can kill the program mutants. We can use these best chromosomes to form four modified specifications for test case generation in the further maintainance of the original program.

To illustrate the efficiency of data generation from the reformed specifications, Table III displays the result of applying the original specifications. For the original specification, we generate test cases 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 defining condition D_i involves unknown output variables that can not directly help to generate test cases.

Compare reformed specifications with the original ones in Figure 5, we can find the $Grade$ of reformed ones are always larger than that of original ones. That means a data suit generated from reformed specifications has 17% higher possibility to pinpoint a bug than that of original ones although both of them share the same $Kill\_rate$ of 100%.

B. Case study 2: gcd

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

Case study 2.

Formal specifications.

TABLE III
RESULTS FOR PROCESS MOD WITH ORIGINAL SPECIFICATIONS

Original specification	Grade	Kill_rate
$T_1 : x > 0 \wedge y \neq 0$	0.43	100%
$T_2 : x < 0 \wedge y \neq 0$	0.40	100%
$\vee_{i=1}^{2} T_i$	0.45	100%

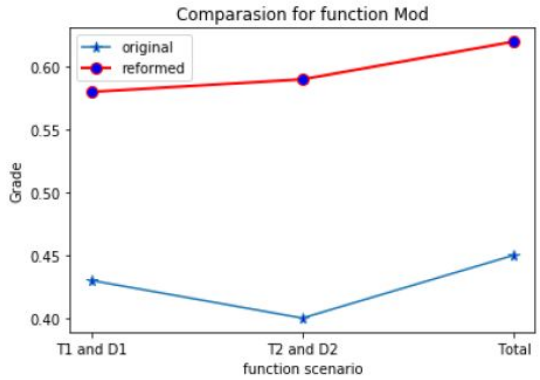

Fig. 5. The Grade of the reformed and original

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, (x - y) / 2)
```

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 IV shows the chromosomes of process *gcd*.

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

The final $Kill\_rate$ of $\vee_{i=1}^{4}[T_i \wedge D_i]_{o'_{i,best}}$ representing the percentage of killed program mutants is 100%. And the corresponding $Grade$ is 0.55, means 55 percent of test cases

TABLE IV
CHROMOSOME FORMS FOR FUNCTION SCENARIOS OF PROCESS GCD

chromosome	d-chromo	dummy vars
$[T_1 \wedge D_1]_{o'}: x \geq y \wedge x\%r \leq d_1 \wedge y\%r \leq d_2 \wedge x\%y\%r \leq d_3$	$o' = (r, d_1, d_2, d_3)$	d_1, d_2, d_3
$[T_2 \wedge D_2]_{o'}: x < y \wedge x\%r \leq d_1 \wedge y\%r \leq d_2 \wedge y\%x\%r \leq d_3$	$o' = (r, d_1, d_2, d_3)$	d_1, d_2, d_3
$[T_3 \wedge D_3]_{o'}: y = 0 \wedge x - r \leq d_1$	$o' = (r, d_1)$	d_1
$[T_4 \wedge D_4]_{o'}: x = 0 \wedge y - r \leq d_1$	$o' = (r, d_1)$	d_1

TABLE V
RESULTS FOR PROCESS GCD AFTER APPLYING GA

best chromosome	Grade	Kill_rate
$[T_1 \wedge D_1]_{o'}$ $o' = (r, d_1, d_2, d_3)$ $o'_{1,best} = (2, 5, 2, 7)$	0.91	100%
$[T_2 \wedge D_2]_{o'}$ $o' = (r, d_1, d_2, d_3)$ $o'_{2,best} = (5, 4, 2, 4)$	0.91	100%
$[T_3 \wedge D_3]_{o'}$ $o' = (r, d_1,)$ $o'_{3,best} = (0, 5)$	0.02	13.3%
$[T_4 \wedge D_4]_{o'}$ $o' = (r, d_1,)$ $o'_{4,best} = (0, 5)$	0.004	6.7%
$\vee_{i=1}^{4}[T_i \wedge D_i]_{o'_{i,best}}$	0.55	100%

generated from $\vee_{i=1}^{4}[T_i \wedge D_i]_{o'_{i,best}}$ can kill all the program mutants.

Then to compare with the original specification, Table V gives the result of original ones.

Compare reformed specifications with the original ones in Figure 6, we can find the first two reformed ones $[T_1 \wedge D_1]_{o'}$ and $[T_2 \wedge D_2]_{o'}$ have very high values 0.91 of $Grade$, which are much larger than that of original ones. In addition, the $Kill\_rate$ of a sole $[T_i \wedge D_i]_{o'}$ $(i = 1, 2)$ is sufficient enough to reach 100%. This means a data suit generated from the two reformed specifications has a dramatically higher possibility to pinpoint a bug. For the last two function scenarios $T_3 \wedge D_3$ and $T_4 \wedge D_4$, due to their very simple forms, we can see there are no any improvements of efficiency after applying our algorithm.

TABLE VI
RESULTS FOR PROCESS GCD WITH ORIGINAL SPECIFICATIONS

Original specification	Grade	Kill_rate
$T_1 : x > 0 \wedge y > 0 \wedge x \geq y$	0.56	93.3%
$T_2 : x > 0 \wedge y > 0 \wedge x < y$	0.63	93.3%
$T_3 : x \geq 0 \wedge y = 0$	0.02	13.3%
$T_4 : y \geq 0 \wedge x = 0$	0.004	6.7%
$\vee_{i=1}^{2} T_i$	0.42	100%

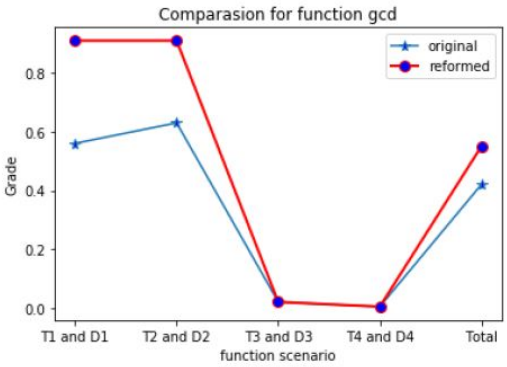

Fig. 6. The Grade of the reformed and original

VI. Conclusion

We propose a new method for efficient test data generation based on reformed pre-post style formal specifications. 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 have also carried out two cases for the evaluation of our method. The result of case studies show that for a complicated function scenario, the proposed method is able to help efficiently generate useful test data to kill as many as possible program mutants.

In our algorithm, by assigning appropriate values to the unknown output variables and by making variations for the original specifications, we can obtain useful reformed specifications that are sensitive to small syntactic structural changes of program codes. The method mainly cope with unknown outputs from formal specifications so as to determine how input data can be generated to test.

However, 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 genetic algorithm usually iterates many times and execute program mutants every iteration, the cost would be not low. But 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 softwares.

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 do more experiments to demonstrate our method can be well used in different kinds of programs.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 26240008.

References

[1] W Eric Wong, Joseph R Horgan, Saul London, and Hiralal Agrawal. A study of effective regression testing in practice. In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*, pages 264–274. IEEE, 1997.

[2] Hareton KN Leung and Lee White. Insights into regression testing (software testing). In *Software Maintenance, 1989., Proceedings., Conference on*, pages 60–69. IEEE, 1989.

[3] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on software Engineering*, (11):777–793, 1996.

[4] Debra Richardson, Owen O'Malley, and Cindy Tittle. *Approaches to specification-based testing*, volume 14. ACM, 1989.

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

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

[7] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *International Symposium of Formal Methods Europe*, pages 268–284. Springer, 1993.

[8] Cliff B Jones. *Systematic software development using VDM*, volume 2. Citeseer, 1990.

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

[10] 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.

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

[12] Yuji Sato and Taku 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.

[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] 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.

[15] 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.

[16] Stuart C Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 64–73. IEEE, 1997.

[17] Elaine J. Weyuker. More experience with data flow testing. *IEEE transactions on software engineering*, 19(9):912–919, 1993.

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

[19] Eliane Martins, Selma B Sabião, 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.

[20] Y. Chen, S. Liu, and F. Nagoya. An Approach to Integration Testing Based on Data Flow Specifications. In *Proceedings of First International Colloquium on Theoretical Aspects of Computing, LNCS*, pages 235–249, Guiyiang, China, September 20-24 2004. Springer-Verlag.

[21] Vadim Okun. *Specification mutation for test generation and analysis*. PhD thesis, Citeseer, 2004.

[22] John Henry Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[23] 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.

[24] S. Liu. *Formal Engineering for Industrial Software Development Using the SOFL Method*. Springer-Verlag, ISBN 3-540-20602-7, 2004.