Journal of Computer Science and Cybernetics, V.33, N.1 (2017), 50–69
DOI 10.15625/1813-9663/33/1/9315

AN IMPROVED GENETIC ALGORITHM FOR TEST DATA GENERATION FOR SIMULINK MODELS

LE THI MY HANH, NGUYEN THANH BINH, KHUAT THANH TUNG

The University of Danang - University of Science and Technology, Vietnam
ltmhanh@dut.udn.vn; ntbinh@dut.udn.vn; thanhtung09t2@gmail.com

Abstract. Mutation testing is a powerful and effective software testing technique to assess the quality of test suites. Although many research works have been done in the field of search-based testing, automatic test data generation based on the mutation analysis method is not straightforward. In this paper, an Improved Genetic Algorithm (IGA) is proposed to increase the quality of test data based on mutation coverage criterion. This algorithm involves some modifications of genetic operators and the employment of memory mechanism to enhance its effectiveness. The proposed approach is implemented to generate test data for Simulink models. The obtained results indicated that IGA outperformed the conventional genetic algorithm in terms of the quality of test sets, and the execution time.

Keywords. Genetic algorithm, mutation testing, simulink, test data generation.

1. INTRODUCTION

Software testing is an expensive, tedious, and time-consuming activity but it is a crucial step to improve the quality and the reliability of software. The process of generating test data decides the effectiveness and efficiency of software testing. The quality of test data is normally measured by the adequacy criteria. Adequacy criteria, also referred to as coverage criteria, pose certain requirements that should be fulfilled by test cases.

Mutation testing proposed by DeMillo *et al.* [6] is a powerful and effective testing technique to assess the quality of test suites. The driving principle of mutation testing is the use of faults which mimic mistakes that a competent programmer would make. These faults are introduced into the program under test once at a time by using mutation operators. The purpose of injecting mutants into programs is both to guide the generation of test data to reveal the faults, and to assess the quality of test data. A test suite is considered good if it contains tests that are able to distinguish a large number of mutants from the original design. If a mutant can be distinguished by at least one of the test cases in the test set, the mutant is considered to be killed. Otherwise, the mutant is alive. The amount of coverage is usually written as the ratio of the number of killed mutants to the entire number of non-equivalent mutants, and called mutation score.

Model-based design is a development methodology for modern software engineering. High level models such as Simulink are widely used to reduce the time of software development in many industrial fields. This also allows faults to be detected at the earlier stages. Verification and validation of Simulink models are becoming vital to users. Therefore, automated test data generation for such models plays a crucial role in practice.

One of the major difficulties in software testing is the automatic generation of test data that satisfy a given adequacy criterion. Generating test cases automatically will reduce cost and efforts significantly. In the recent work as presented in [17], the genetic algorithm was employed to generate test data satisfying mutation coverage. Though the results were promising, the algorithm could be enhanced. This paper proposes an Improved Genetic Algorithm with the alteration of genetic operators by using a multi-parent crossover method and a novel population diversity operator as well as employing a memory mechanism to store the potential individuals which are able to kill mutants. The IGA aims to make an improvement of the original GA in terms of the obtained mutation score, execution time, and the number of executed mutants for the problem of test data generation. To our knowledge, this is the first work using the genetic algorithm with multi-parent crossover incorporating with the memory set mechanism to generate test input data based on mutation testing.

The rest of this paper is organized as follows. The definitions of mutation testing, Simulink and mutation operators for Simulink models are presented in Section 2. Background of genetic algorithms, and the techniques for generating test data based on these algorithms are described in Section 3. Section 4 proposes an Improved Genetic Algorithm for the generation of test data. The case studies and results are reported and discussed in Section 5. Conclusions and future works are presented in Section 6.

2. BACKGROUND

2.1. Simulink

Simulink [44] tool set is integrated into MATLAB and is used for modeling, and simulating dynamic systems. Simulink models are widely utilized in the design and implementation of embedded systems, comprising electronics, electrics, aerospace, and automobile systems. They include functional blocks with input and output ports interconnected via lines. These interconnections indicate the data flows among the blocks, and set up equations relating the involved interface variables. Blocks are built-in functional units utilized to produce, manipulate, and output signals. A block can be a parent container containing other blocks known as the subsystem or the sub-functionality. Each block has a number of parameters, for instance, initial values and ranges, which control the operation of the block. Lines supply a mechanism to transmit data from inputs to outputs of a model. Fig. 1 shows an example of a Simulink model.

2.2. Mutation testing for Simulink models

Mutation testing [6, 14] is an efficient software testing approach being able to simulate software defects systematically and determine the quality of a test set. These defects are called mutants, and they are generated by simple syntactic rules such as alterations of operators, constant values, data type, and variables. These rules are called mutation operators. These operators can be seen as representing common faults usually found in software. Therefore, they are designed referring to the experience of using the target language and the most common faults.

We get a faulty model, which is called mutant, when applying a mutation operator to a model. Test data then should be generated to reveal the mutants. Mutants are killed

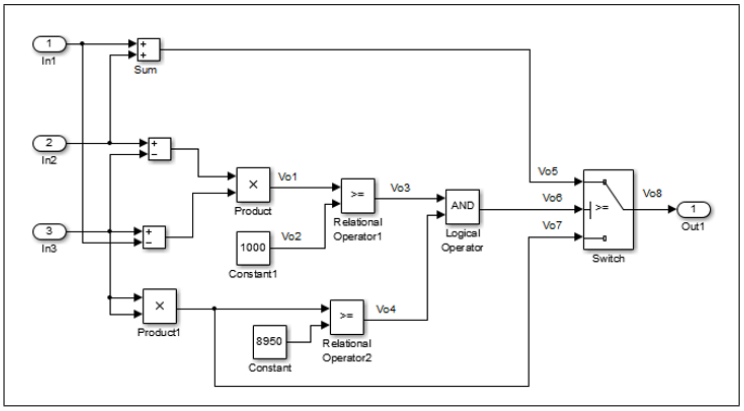

Figure 1. A Simulink model

when their outputs differ from those of the original program. Otherwise, mutants are called alive. Generally, there are two possible reasons why a mutant has been left alive. The first one is that the used test cases are not capable of revealing fault, whilst the second one is that there is no such data. In the second case, the mutant model is functionally equivalent to the original one, and called the equivalent mutant. Worse still, determining whether a mutant is equivalent is undecidable [44], and so typically the decision is left for testers to establish manually. After executing a large set of test data against mutants, there are still several mutants that are not killed by any test data in the test suite but they are not equivalent mutants. These mutants are seen as stubborn mutants [43], also known as hard-to-kill mutants. Test data which are able to kill such mutants locate in the extremely narrow range of input data. This means that the test suite is not yet adequate, and it needs the enhancement. In contrast to hard-to-kill mutants, weak mutants [22] are killed by every test case.

Mutation operators are the pivotal factor in mutation testing. In order to apply mutation testing to Simulink models, faults are systematically inserted into models, and then mutant and original models are executed on test data in order to observe how many of faults are discovered. The more the faults can be revealed, the better test set is. In our approach, errors are introduced into the system by perturbing the values of signals carried on wires/lines as well as changing the operations performed within blocks. Table 1 presents the set of mutation operators for Simulink models that were proposed in [15].

3. TEST DATA GENERATION USING GENETIC ALGORITHM

The generation of test data is one of the most important steps in software testing. This activity is normally conducted manually. However, manual test data generation is a hard, laborious, and very time-consuming task. Automating this step in testing can greatly reduce the human effort and cost.

The test data generator aims to find test cases that satisfy a given coverage criterion. Structural coverage and mutation coverage are commonly used as adequacy criteria to generate test data. There are some prominent techniques, which automate the process of gen-

Table 1. Mutation operators for Simulink

Operator	Description
VNO	Variable Negation Operator
VCO	Variable Change Operator
CCO	Constant Change Operator
CRO	Constant Replacement Operator
SCO	Statement Change Operator
SSO	Statement Swap Operator
ROR	Relational Operator Replacement Operator
AOR	Arithmetic Operator Replacement Operator
ASR	Arithmetic Sign Replacement Operator
LOR	Logical Operator Replacement Operator
BRO	Block Removal Operator
SRO	Subsystem Replacement Operator

erating test data. The most popular ones are random generation of test data [5], symbolic execution [45, 31], search-based generation of test data [26], and recently generation of test data based on genetic algorithm [17, 4]. This paper focuses on the last one.

In recent years, there have been a large number of studies [35, 37, 42, 34, 30] focusing on automatic generation of test data. In [37], Vincenzi *et al.* figured out the adequacy, effectiveness, and cost of manually generated test sets versus automatically created test sets for Java programs. They observed that the use of automatic test generators can form test cases which traverse uncovered parts of the source code that manual test cases cannot do. Authors also showed that the combination of manual and automated test sets might increase statement coverage and mutation score more than 10% on average in comparison with only using manual test sets. In addition, one drawback of manual test data generation is very costly. Therefore, automatic generation of test data is highly recommended. In [34], Singh addressed the meaning and usefulness of automated test data generation. He reviewed the practical applicability of existing automated methods of expected output generation. Finally, he claimed that the existing work is very basic and preliminary in nature. Therefore, the demand for generating expected outputs is extremely essential to support for automated testing. Mutation analysis is an effective approach to assess the performance of test sets based on the ability to find faults. These things motivate us to apply mutation criteria to create an automatic tools for generating the efficient test data for Simulink models. To support for providing useful test cases, Xu *et al.* [42] proposed an adaptive fitness function based on branch hardness. That work showed its effectiveness and efficiency compared to similar studies. However, the authors did not show how their approach might be automated and to which field of applications it can be applied. In contrast, we study a meta-heuristic algorithm to generate test data automatically. Besides to branch coverage, mutation criteria is also an effective manner to generate test sets. In [35], Souza *et al.* introduced an automated test generation approach for strong mutation using hill climbing. In general, empirical results on 18 C programs showed the low mutation scores because hill climbing is a local optimal algorithm. Meanwhile, our work aims to utilize a global optimization technique, i.e. genetic algorithm, to obtain better mutation scores. Another research carried out by Paduraru *et al.* [30] presented a parallel implementation of a genetic algorithm in Apache Spark to reduce time for test data generation for executable programs with a given

fitness function. Their method of fitness evaluation is based on some "probabilities" that certain branch conditions occur in some order. This means that their approach is only used for structural testing with no concentration on improving the ability to fault discovery of test inputs. Meanwhile, other literature such as [28] pointed out that mutation testing is shown to subsume most structural criteria, and it is more effective than structural testing. Hence, the goal of our study is to combine an improve genetic algorithm with mutation criteria aiming to generate test sets that are more likely to uncover potential faults in Simulink models.

Genetic algorithm (GA) [18] is one of the most popular meta-heuristic search algorithms. It mimics the evolutionary processes in nature: a population of initially randomly generated candidate solutions is evolved using genetic operations such as crossover, mutation, and selection. The evolutions are guided by a fitness function that heuristically measures how good a candidate solution is. The fitness of the individuals would be gradually improved through generations, and the search process stops when an optimal solution has been found, or when some other predefined stopping conditions (e.g. the maximum number of generations or fitness evaluations) have been met.

Using the genetic algorithm in conjunction with structural coverage has received much attention in research. Xanthakis *et al.* [41] presented the first work applying the genetic algorithm to generate test data. In their work, GA is used to generate test data for structures not covered by the random search method. A path is chosen by the user, and the relevant branch predicates are extracted from the program. The GA is then used to find input data that satisfy all branch predicates at once, with the fitness function summing branch distance values. In [32], Peng and Lu used the user session data in their request dependence graph to generate test cases by applying GA. Girgis [10] proposed a structural oriented automatic test data generation technique that uses the GA guided by the data flow dependencies in the program to carry out the all-uses criterion. The program under test is converted into a Control Flow Graph where each node represents a block in the program, and the edges of the flow graph depict the control flow of the statements. Roper *et al.* [33] developed a system to automatically generate test data to achieve a high level of coverage for C programs using the GA. The system creates an initial population of random data based on a description of the input data then performs an iterative search based on the GA, which involves running this data and measuring its coverage. The GA is not only used to generate data for the code level structural testing but also applied for high level models. Derderian *et al.* [8] applied the GA to generate test data for the Finite State Machine (FSM) with temporal constraints. The fitness function was based on the number of temporal constraint violations committed by each candidate input sequence. Lefticaru and Ipate [20] proposed an application of the GA to generate test input that is executed along a specific path in an extended FSM, drawn from UML models. Windisch [38] adopted the GA in order to generate the continuous input signal for real-time Simulink/Stateflow models based on the branch coverage criterion. Ghani *et al.* [9] used the GA and the Simulated Annealing (SA) for the Switch block path coverage of Simulink models. They reported that the GA and SA algorithms achieved similar coverage, but the GA was more often successful. Oh *et al.* [29] have also introduced a messy-GA for the transition coverage of Simulink models.

In addition to structural coverage, mutation coverage is also an effective criterion to support for test data generation. Bottacy [4] proposed a fitness function for genetic algorithms based on the constraints defined by DeMillo and Offutt [7] to generate mutation-based test

data. Louzada *et al.* [21] proposed the generation and selection of test data evolved by a GA that uses a mutation score as a fitness function. The mutation score is found by running the mutants of the program generated from the benchmarks used. Haga and Suehiro [11] proposed a method that automatically generates test cases based on the genetic algorithm and the mutation analysis for C programs. Their method combines the random generation and refinement. Each test datum is randomly generated in the first step, and then a set of test data is refined by the genetic algorithm. In Hage and Suehiros study, mutation scores were used to measure the adequacy of the test data set. Last *et al.* [19] used the fuzzy based the extension of the GA approach for test case generation. The aim is to find a minimal set of test data that is likely to expose faults using mutated versions of the original program.

In [17], a GA was proposed to generate test data for Simulink models. An individual is considered as a set of tests, and a roulette wheel selection technique [3] is used to select individuals for reproduction. In this approach, individuals are selected with a probability that is directly proportional to their fitness values. The fitness in our work is computed based on the total number of mutants killed by all the tests that belong to an individual. The best individual is retained in each generation before performing crossover and mutation. After performing the double point crossover, a mutation operator will be done with the pre-defined mutation probability. For each locus mutated, the test data in this locus are replaced by other ones. After a number of generations, the algorithm will return the best individual of population which kills the most mutants. This approach has some limitations such as requiring much memory, and spending a lot of time generating test data as well as the quality of test data being not high.

When applying the GA to the generation of test data based on mutation testing, the authors in [2], and [25] represented each individual as a set of tests. This representation faces some restrictions as described above. This paper proposes an improved GA with some modifications in the individual representation to overcome these restrictions. Most studies of test data generation applying the conventional GA in [9, 41, 11, 2, 25, 24, 36] used single crossover to create two offspring from two selected parents. With this crossover, offspring are not quite different from their parents because the genetic materials of the parents are passed on to the offspring. If the test data are distributed in a wide space, it takes a long time to find out the right test data killing the mutants. A new three-parent crossover method is employed in this paper to generate three new offspring to improve the effectiveness of the GA. A new diversity operator is also proposed to enable the algorithm to escape from the local optima. It replaces the existing mutation approach which simply chooses a gene and replaces it randomly as in [9, 17, 41, 11, 2, 25, 24, 36]. The details of the approach will be presented in the next section.

4. THE IMPROVED GENETIC ALGORITHM FOR TEST DATA GENERATION

Genetic Algorithms, which were proposed in [17, 2, 25], represented each individual as a set of test data. In each genetic generation, these test sets have to be executed on the entire of mutants in order to specify the fitness of each individual. This representation uses a lot of memory and increases the execution time of the algorithm. From this analysis, this paper proposes the Improved Genetic Algorithm (IGA), which is the combination of the

conventional GA and the memory mechanism. In mutation testing, no single test case can kill all mutants, so a good solution is a set of tests killing the most mutants. Therefore, a memory set is employed in order to achieve a good individual through genetic generations. The employment of the memory concept for the conventional GA has led to the change of the individual representation and genetic operators such as crossover and mutation.

4.1. Mutation-based test data generation problem

The statement of the mutation-based test data generation problem is as follows.
Suppose:

- MUT is the Simulink model under test;
- V is the number of inputs of MUT;
- $\overrightarrow{X} = \{x_1, x_2, ..., x_v\}$ is a test datum of MUT.

So that, $L_k \leq x_k \leq U_k, x_k \in \mathbb{R}(k \in [1, V])$.
where L_k is a lower limit, and U_k is an upper limit of the input x_k, R is a set of real numbers.
We need to find

$$S = \{\overrightarrow{x_1}, \overrightarrow{x_2}, ..., \overrightarrow{x_v}\},$$

so that $Killed(S)$ is maximal, and n is minimal in G genetic generations of the IGA, where S is the memory set that contains the individuals which are able to kill mutants, and $Killed(S)$ is the number of killed mutants by S. In other words, mutation-based test data generation is a process that injects syntactic mutations into the model under test MUT, and then generates from the MUT a set of mutants M. The purpose of this activity is to seek a set of test cases S that is able to kill most of the mutants in M. This means that the test cases make each mutant $m \in M$ generate outputs that differ from those of the original model MUT. Each test datum in S is a vector of input data whose the dimensionality is equal to the number of inputs of MUT. In order to reduce the execution time of the mutation testing process, the number of test data in S must be optimized as much as possible.

Our approach represents each individual in the IGA by a test datum $\overrightarrow{X}$ instead of a set of tests. This representation uses less memory than the GA in [2, 25]. For each genetic generation, a good solution will be added to S. A good solution to the problem is a test datum which can kill at least one mutant which was not killed by any other solutions in S. After a number of genetic generations, S will be returned as the set of test data which is able to kill as many mutants as possible.

4.2. Fitness function for mutation-based test data generation

In order to select the good individuals to reproduce, the roulette wheel selection technique [3] is used based on their fitness. This paper proposes a new fitness function using the number of "hard-to-kill" mutants that each individual killed in the current generation. As the IGA uses a memory set to archive the best individuals in each genetic generation, the fitness function for each individual in the current population is computed based on the number of alive mutants in the current generation.

Table 2. Example of computing fitness function

	M_1	M_2	M_3	M_4	M_5	f
X_1	1	0	1	0	1	3.699
X_2	1	1	1	0	0	3.504
X_3	0	0	1	0	1	2.613
X_4	1	1	0	1	0	5.102
X_5	0	1	0	0	1	2.613
X_6	1	1	0	0	0	2.295
X_7	1	0	1	0	0	2.295
δ_m	1.086	1.209	1.209	2.807	1.404	

$$f(\overrightarrow{X_i}) = \sum_{m=1}^{M} (\alpha_m \cdot \delta_m), \tag{1}$$

$$\alpha_m = \begin{cases} 1, & \text{if } \overrightarrow{X}_i \text{ kills mutant } m \\ 0, & \text{else,} \end{cases} \tag{2}$$

$$\delta_m = \begin{cases} 0, & NoT_m = 0 \\ log_{(NoT_m)}T, & \text{else,} \end{cases} \tag{3}$$

where T is the number of individuals in the current population, M is the number of alive mutants in the current genetic generation, NoT_m is the number of individuals that can detect mutant m. $\overrightarrow{X_i}$ is the i^{th} individual in the population, and f is the fitness function.

An example of computing the fitness function is illustrated in Table 2. Given 5 alive mutant $M_1, ..., M_5$ of the current genetic generation having 7 individuals $X_1, ..., X_7$, if mutant M_j is killed by individual X_i, then the element at row i, and column j is 1. Otherwise, it is 0.

Individuals with higher value of f have more chances to be selected to generate the new generation. In this example, X_4, X_1, and X_2 are more likely to be selected than the remaining individuals.

4.3. A multi-parent crossover operator for the improved genetic algorithm

In this paper, the multi-parent crossover (MPC) is proposed with the following steps:

1. Using the Roulette Wheel Selection algorithm to select three distinct individuals in the current population;

2. Ranking these three individuals from the best $\overrightarrow{X}_3$ to the worst $\overrightarrow{X}_1$ based on their fitness functions;

3. Generating a random number α from the standard uniform distribution in the interval of [0, 1];

4. Generating three offspring:

$$\overrightarrow{O}_1 = \overrightarrow{X}_1 + \alpha \cdot (\overrightarrow{X}_2 - \overrightarrow{X}_3), \tag{4}$$

$$\overrightarrow{O}_2 = \overrightarrow{X}_2 + \alpha \cdot (\overrightarrow{X}_3 - \overrightarrow{X}_1), \tag{5}$$

$$\overrightarrow{O}_3 = \overrightarrow{X}_3 + \alpha \cdot (\overrightarrow{X}_1 - \overrightarrow{X}_2), \tag{6}$$

where $f(\overrightarrow{X}_1) \leq f(\overrightarrow{X}_2) \leq f(\overrightarrow{X}_3)$.

In [40], Wright proposed a heuristic crossover, in which one offspring $\overrightarrow{O}$ is generated from two parent individuals $\overrightarrow{X}_1$ and $\overrightarrow{X}_2$, such that $\overrightarrow{O} = \overrightarrow{X}_1 + rand \cdot \overrightarrow{X}_2$, where *rand* is a number between 0 and 1, and $f(\overrightarrow{X}_1) \leq f(\overrightarrow{X}_2)$. This crossover uses two parents. Our approach is inspired by Wrights heuristic crossover but three parents are used instead of two. Equations 4 and 6 are designed to move towards better fitness for child individuals while Equation 5 is to diversify the population.

4.4. A mutation operator for the improved genetic algorithm

In addition to the crossover operation, the mutation operator [18] used to maintain genetic diversity from one generation of a population of individuals to the next. The mutation operator helps to prevent the premature convergence of population in the GA as well. The mutation of an individual strongly depends on the choice of representation for an individual. In [23], Malhotra *et al.* described some techniques that were used to create the mutation for each individual such as binary encoding mutation, permutation encoding mutation, value encoding mutation, and tree encoding mutation. As mentioned above, each individual in this work is a real-number vector of test data for the Simulink model under test. In this section, a new mutation operator in real encoding for the genetic algorithm is proposed to enhance the diversity of population. This operator is built by combining information from generated offspring, individuals in the current memory set S and individuals in the parent population P. Suppose that MR is mutation probability, and $O = \{o_1, o_2, ..., o_j, ..., o_V\}$ is the mutated offspring. The individual O is then mutated as follows:

$\forall i \in \{1..V\}$, V is the number of inputs of Simulink model, and x is a uniform random number in [0, 1]. If $x < MR$ then

$$o_i = \begin{cases} 0, & z < 0.1 \\ y \cdot o_i + (1 - y) \cdot p_i^k, & y = rand(0, 1), 0.1 \leq z < 0.5 \\ s_i^k, & 0.5 \leq z \leq 1 \end{cases} \tag{7}$$

where z is a random number in the range of [0, 1], $P^k = \{p_1^k, p_2^k, ..., p_V^k\}$ is a randomly selected individual in the parent population P; $S^k = \{s_1^k, s_2^k, ..., s_V^k\}$ is a randomly selected individual in the current memory set S.

From the experiments in our previous work, it is recognized that some mutants of Simulink models are detected when the input value is 0. It is difficult for inputs to get the value of 0 because input spaces are vast, and test data are randomly generated in the initial population. For the mutation operator of the IGA, thus, some input values are assigned to 0 with a given probability. The individuals in the memory set are the best individuals that kill the mutants of the Simulink model in the current genetic generation. It is expected that the use of information of these individuals for the mutated individuals will create better individuals. The values of parent individuals are also used to increase the diversity of child population. The diversity of the mutation operator will help the IGA avoid getting stuck in a local minimum.

4.5. The improved genetic algorithm for test data generation for Simulink models

Algorithm 1 presents the IGA for test data generation for Simulink models. In the IGA, first an initial population is randomly generated with *popSize* individuals (*line 2*). Then, the fitness of each individual in the current population is computed using equation 1 (*line 4*). The best individuals in the current population are added to S (*line 5-6*). These individuals can kill at least one mutant which was not killed by any other individuals in S. Next, three distinct individuals are selected respectively (*line 11-13*), and the crossover and mutation operators as described above are applied before creating three offspring until the size of child population is equal to *popSize* (*line 14-18*). After that, the fitness of individuals in the child population is computed (*line 20*), and the best individuals are

Algorithm 1 The IGA for test data generation for Simulink models

Require: Initial population size (*popSize*), number of generations (*numGen*), mutation rate (*MR*).
Ensure: A set of test data for the Simulink model under test.

```
1: begin
2: population ← InitPopulation( popSize);
3: S ← ∅ //Initialize the empty memory set to store the best individuals
4: ComputeFitness(population);
5: BestInd ← SelectBestIndividual(population);
6: ArchiveBestIndividual(BestInd, S);
7: g ← 0
8: while g < numGen do
9:     popChild ← ∅ ; // Create an empty child population
10:    while size(popChild) < popSize do
11:        I1 ← Select(population);
12:        I2 ← Select(population);
13:        I3 ← Select(population); //I1 ≠ I2 ≠ I3
14:        MultiParentCrossover(I1, I2, I3; O1, O2, O3);
15:        O1 ← Mutate(O1, MR);
16:        O2 ← Mutate(O2, MR);
17:        O3 ← Mutate(O3, MR);
18:        popChild ← Combine(O1, O2, O3, popChild);
19:    end while
20:    ComputeFitness(popChild);
21:    BestInd ← SelectBestIndividual(popChild);
22:    ArchiveBestIndividual(BestInd, S);
23:    population ← popChild;
24:    g = g + 1; // Next genetic generation
25: end while
26: S ← Optimize(S);
27: return S;
28: end
```

added to memory set S (*line 21-22*). The child population then becomes a new population for the subsequent evolution. The IGA continues until the termination criterion such as the maximal number of generations is met. Finally, a set of the best individuals will be returned after being optimized (*line 26-27*). The method to optimize the memory set of best individuals is presented in the next subsection.

4.6. Optimizing memory set

It is found that the final set of all the memorized test data may not be minimal since the algorithm only saves the best individuals of one generation, and it omits information to guide the process of minimizing the number of test data in the memory set through the generations. The minimization can be done in a separate phase once the IGA finishes. A boolean matrix, called A, is built with rows being test data, and columns being mutants. $A_{ij} = 1$ means that the i^{th} test data kills the j^{th} mutant, and $A_{ij} = 0$ means that it does not. Then, this matrix is optimized by reusing the algorithm proposed in [16] in order to create the minimal test set.

5. EXPERIMENTATION

The proposed approach is implemented in the MuSimulink tool [13] to generate test data for Simulink models. Four models from [9] and the Quadratic_v2 model from [44] are used to assess our approach. The experiments are conducted on a PC AMD Operon Dual-Core 2.27 GHz with 4 GB memory, running the Windows Server 2008 operating system. Mutants for each model are generated by using the set of mutation operators in [15] presented in Table 1.

5.1. Research questions

The aim of our experimental work is to assess the effectiveness of the IGA in comparison with the GA, Artificial Immune System (AIS), and Simulated Annealing (SA) algorithms as shown in [12] in terms of criteria as follows:

- How the mutation score can be enhanced by using the IGA. The higher mutation score is, the better algorithm is.
- How the time of generating test data is reduced when employing the IGA compared with the conventional GA, SA, and AIS algorithms. With the same mutation score, which algorithm gives higher mutation score in less runtime will be better.
- How the effectiveness of using the memory set mechanism can contribute to reducing the number of executed mutants.

The number of mutants executed against test data for each algorithm in order to achieve an expected mutation score reflects the efficiency of algorithms in reducing the execution time. Mutation testing is a computationally expensive testing technique. The most expensive computation parts of the mutation process are original execution, mutant execution, and output comparison for each test input [27]. Therefore, if the number of executed mutants is reduced then the execution time will decrease significantly. This criterion is used to assess the efficiency of the IGA and the GA to highlight the advantages of the improved tasks.

5.2. Experiment parameters

Each algorithm has its own parameters that influence its performance in terms of the solution quality and the processing time. The population size is a pivotal parameter which have a great effect on the effectiveness of all population-based algorithms such as the genetic algorithm. To obtain the most suitable parameter values that suit the test problems, different settings were experimented. The parameter values were changed one by one, and the results were monitored in terms of the solution quality based on the mutation score and the execution time. As a result, the most appropriate settings of the population size parameter $popSize = 1000$, and the number of genetic generations $numGen = 10$ for the IGA were considered during the experiments. As mentioned above, each individual in the original GA is represented as a set of test cases, thus in order to ensure algorithms to be comparable with each other in terms of the number of executed mutants, the number of tests in the initial population must be equal. While the size of population of the IGA had been 1000, and each individual of the IGA had been a test datum, extensive experiments were run in order to determine the most appropriate settings of the population size, and the number of test cases per each individual for the GA in order to the multiplication of these two parameters was 1000. After experimenting with various values, a population size of 25 individuals, and an individual size of 40 test cases were used for the GA. The similar experiments are also carried out with the AIS and the SA following the parameters of each algorithm presented in [12]. The final parameter values adopted for each of algorithms are given as follows.

GA Configuration The following is a summary of our experiments parameters.

- The initial population size: 25
- The number of test cases per each individual: 40
- The number of generations: 10
- The crossover rate: 0.9
- The mutation rate: 0.5

Table 3. The obtained mutation score by algorithms

Model	No. of mutants	Mutation Score (%)			
		IGA	GA	AIS	SA
SmplSw	92	95.65	95.65	**96.74**	94.57
Quadratic_v1	161	**88.82**	86.33	**88.82**	87.58
RandMdl	188	**97.34**	86.17	94.68	88.30
Tiny	144	**93.06**	84.72	**93.06**	88.89
Quadratic_v2	140	**82.14**	75.71	80.57	75.71

SA Configuration The SA has some parameters as follows:

- The value of temperature reduction function: 0.95
- The initial temperature:1000
- The number of test data per solution: 1000
- The maximal number of iterations: 10

AIS Configuration The AIS depends on some configuration parameters as below:

- The number of iterations: 10
- The size of population: 1000
- The number of best individuals that are selected to proliferate: 40
- The user-defined parameter about the number of clones created for each selected member: 4
- The number of new randomly generated antibodies to replace the lowest affinity antibodies in the population: 40

IGA Configuration The IGA depends on the following parameters:

- The initial population size: 1000
- The number of generations: 10
- The mutation rate: 0.1

5.3. Experiment 1

This experiment aims to show the superiority of the IGA compared with other algorithms including conventional GA, SA, and AIS.

Each model was executed at least 10 times per each algorithm to obtain statistically significant results. For each execution, the algorithms were performed with the same input domain description for the Simulink model under test. The same parameters including the number of iterations, and the number of tests in the initial population are used for four algorithms. The number of tests in each genetic generation is balanced. The most appropriate measures of the algorithm effectiveness are the time taken to generate test data and the mutation score achieved. A more effective algorithm will achieve a higher mutation score in less time. The results, which are presented in Table 3, are the averaged mutation scores within 10 execution times for each model.

Table 4. The execution time and the total number of executed mutants on experiments

Model	Time (s)				No. of executed mutants			
	IGA	GA	AIS	SA	IGA	GA	AIS	SA
SmplSw	7790.849	9042.871	**6850.845**	9160.547	132000	181730	**120000**	181890
Quadratic_v1	14457.963	15258.735	**12872.125**	15129.236	341000	370229	**327000**	367580
RandMdl	**12743.943**	16691.729	13185.634	15925.423	**302000**	483192	315000	454690
Tiny	11028.495	16904.719	**10230.589**	14690.482	253000	378297	**246580**	361630
Quadratic_v2	**21131.584**	23706.108	21290.543	23547.361	**428000**	560998	430750	542820

It can be observed that the IGA is stable, and it has significantly improved the number of killed mutants compared to the GA and the SA. Generally, the IGA kills on average approximately 6% and 4% more mutants than the GA and SA do respectively. The average mutation score by using the IGA is slightly higher than that of using the AIS (about 0.6%). In particular, the IGA kills more mutants than the GA and the SA do in all models, while the number of killed mutants by using the IGA is higher than that of using the AIS in two models *Quadratic_v2* and *RandMdl*. It also can be seen that the number of killed mutants is the same when adopting the IGA and the AIS in two models *Tiny* and *Quadratic_v1*. By using the manual analysis, we find that both models *RandMdl* and *Quadratic_v2* are complicated with many hard-to-kill mutants. This points out that the IGA is an effective algorithm compared with the other algorithms.

In the IGA and AIS algorithms, the memory set is employed to archive the best individuals in each genetic generation, and this memory set is automatically evolved through generations. Therefore, we do not need to initialize the number of test data for the memory set as we did for each individual in the GA and the SA. This mechanism contributes to the increase of the mutation score for the model under test. Other reasons which explain the effectiveness of the IGA are the use of the new fitness function, the multiple parent crossover and the new mutation operator compared to the fitness function using mutation score in the GA, AIS, and SA algorithms in previous works.

Table 4 presents the execution time and the total number of executed mutants for each algorithm on case studies within 10 genetic generations.

As for the number of executed mutants, the individuals of the GA and SA algorithms have to be executed against the entire number of generated mutants of the model. In contrast, for each genetic generation of the IGA and AIS algorithms, test cases are executed against only mutants that are alive in the current generation by maintaining a memory set to save test data being able to kill mutants as well as eliminating the killed mutants in previous genetic generations. Thus, the total number of executed mutants, and the execution time of the IGA and AIS algorithms are always less than those of the conventional GA and SA ones. In addition, the number of executed mutants also depends on the nature of algorithms and the parameter configurations, so the number of executed mutants by using the IGA is lower than the figure for the AIS in several models such as *RandMdl* and *Quadratic_v2*, while it is higher than that of using the AIS in other models. Fig. 2 presents the total number of executed mutants of four algorithms.

In terms of the execution time, the IGA is faster than the conventional genetic algorithm and the SA. However, the execution time of the IGA is less than that of the AIS in only two models *Quadratic_v2* and *RandMdl*. In the test data generation process, all algorithms must execute mutants to compute the mutation score. The number of executed mutants reflects the time the algorithm takes. The number of executed mutants shown in Table 4 indicates its relationship with the execution time. The fewer number of executed mutants is, the less execution time is.

5.4. Experiment 2

This experiment assesses the performance of the IGA compared to the conventional GA to point out the efficiency of the improved algorithm version in terms of the number of executed mutants for each algorithm. In experiment 1, both models *RandMdl* and *Quadratic_v2* are complicated, and they

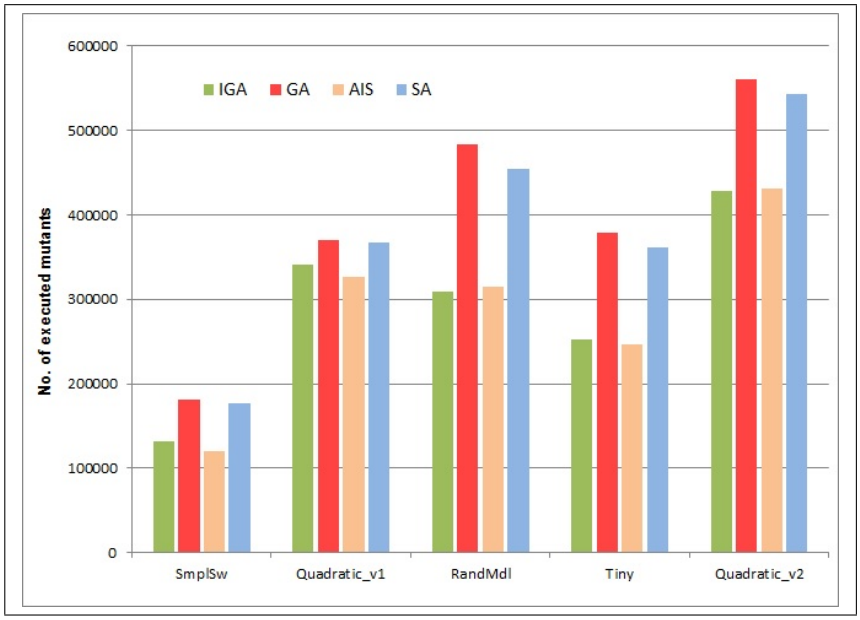

Figure 2. Comparing the total number of executed mutants of algorithms

generate many hard-to-kill mutants, so the number of generated mutants in both these models is much higher than that of other models when employing the GA. Therefore, we use the *RandMdl* and *Quadratic_v2* models to evaluate the performance of the IGA compared with that of the GA in this experiment. The results for other models are as similar as the typical results illustrated below. Fig. 3 and Fig. 4 show the number of mutants executed to achieve at least a specific mutation score for two Simulink models.

During the initial stages, the number of executed mutants is low for both algorithms. When the mutation score increases, the number of mutants needed to be executed to improve the mutation score increases dramatically for the GA before increasing for the IGA. In early iterations of both algorithms, weak mutants are killed by test data easily resulting in a large increase in the mutation score with few mutants executed. However, in the later genetic generations, only hard-to-kill mutants are remaining. It is difficult to generate test data to kill these mutants. Thus, more mutants are executed before a test being found to improve the mutation score. This is the reason why the number of executed mutants rises rapidly to reach a high mutation score.

In the conventional GA, an individual is represented as follows $I = \{a_1, a_2, ..., a_i, ..., a_n\}$ where a_i is the i^{th} test datum, n is the total number of tests in an individual. When this individual executes against M mutants generated by the model under test, the test datum a_i just executes on the mutants that are not yet killed by the test data from a_1 to a_{i-1}. Therefore, the total number of mutants which are executed by the individual I is fewer than $M \times n$. In the initial generation of the IGA, each test datum must be executed with M mutants. Thus, if the population has got n individuals then $M \times n$ mutants need to be executed. That is why the number of executed mutants in initial generations of the IGA is greater than that of the GA. For the GA, in the subsequent genetic generations, the number of executed mutants of each individual is the same as the initial generation because each test datum of the individual has to execute against M mutants in each generation. In contrast, the number of executed mutants of the IGA decreases through generations because only the alive mutants are executed, and mutants killed by the test data which are stored in memory set are not executed on the test datum of the individual in the current generation. In addition, the weak mutants are considerably killed through initial generations, so the number of alive mutants needs executing on

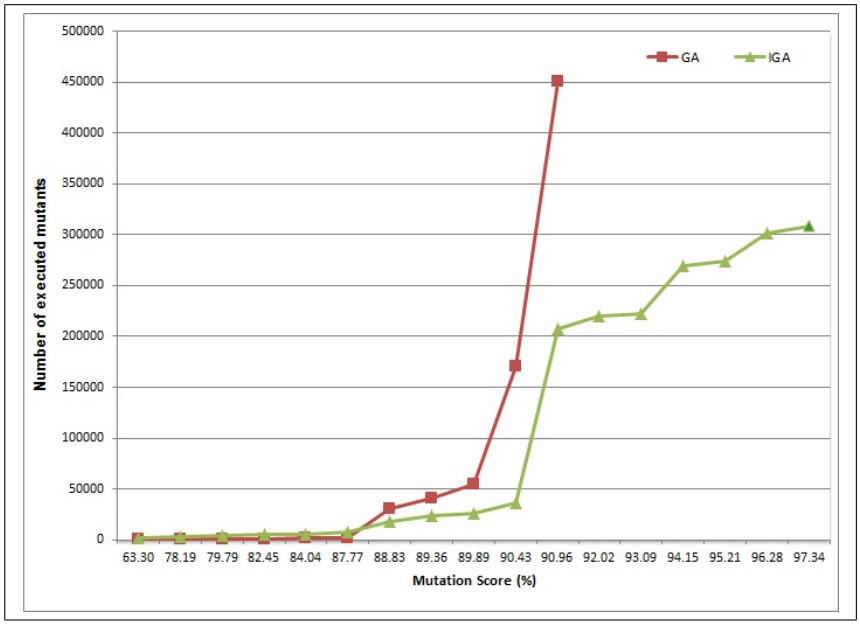

Figure 3. The number of mutants executed to achieve at least a specific mutation score for RandMdl model

the test data will decline significantly in the next generation of the IGA. This is the outstanding feature in the use of the memory set to store the test data that are able to kill mutants and to remove the killed mutants from the set of mutants which needs to be executed in each genetic generation. Therefore, while the total number of executed mutants of the GA increases dramatically, the figure for the IGA rises much more slowly when increasing the number of genetic generations.

The graphs also show that the IGA is able to generate more hard tests in fewer executions, leading to higher mutation scores for the same number of executions. In other words, the IGA needs fewer executions to achieve the same mutation scores. These results show that the IGA is more effective than the GA.

Although the obtained results are promising, the tests cannot kill all generated mutants because of several reasons:

- Equivalent mutants are neither identified nor excluded from the experiments. This is because knowing what mutants are equivalent requires the previous execution of all the mutants, followed by the searching for the equivalent ones among the alive ones.
- In this approach, the fitness function orients to whole mutants, so it does not guide the search process by assessing the distance to the test, which is able to kill specific mutants, from current test. Thereby, if a model has many hard-to-kill mutants, the mutation score is not very high. For these reasons, we continue to study a more effective fitness function in further work.
- Four algorithms IGA, AIS, SA, and GA depend on configuration parameters. Thus, many experiments are necessary to determine the most appropriate combination.

5.5. Threats to validity

The threats to validity should be taken into consideration throughout any empirical study. Wohlin *et al.* [39] discussed four main types of validity threats: conclusion, internal, construct, and external.

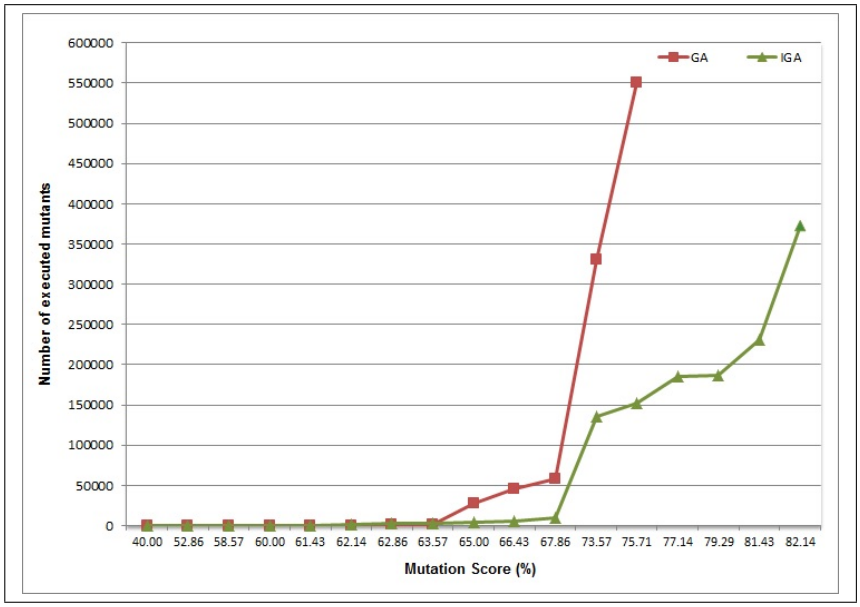

Figure 4. The number of mutants executed to achieve at least a specific mutation score for the Quadratic_v2 model

This paper compares the Improved Genetic Algorithm to the conventional Genetic Algorithm, the SA, and the AIS in terms of the mutation score, the execution time, and the number of executed mutants. Threats to *conclusion validity* focus on how sure we can be that the treatment used in an experiment is really concerned with the actual outcome we observed. Since test data generation techniques use genetic algorithms, and the initial population of these algorithms is usually randomly generated, the experiments may deliver different results. To cope with this problem, we carried out each experiment 10 times, and we followed rigorous statistical procedures to assess the results. Ten runs were a rule-of-thumb limit proposed by Ali *et al.* in [1].

Threats to *construct validity* are on how the performance of a testing technique is defined. We measured the performance of IGA, AIS, SA, and GA algorithms in terms of the mutation score of the obtained test sets. However, this performance measure is hard to manually evaluate the generated test cases (i.e. to check the correctness of the outputs between the original model and the mutant model on test cases), because of the large number of generated mutants from the model under test. The difficulty in the manual assessment of the generated test cases might affect to the accuracy of the obtained results.

Threats to *internal validity* might come from factors that have an impact on the outcome, in particular poor parameter settings. To deal with this problem, many extra-experiments were conducted to choose the appropriate parameters for the algorithms. The complete description of parameter values is presented in the experimental section to help other researchers can reproduce easily. However, due to the time limit, the tests which were run in the experiments may not be long enough. This can result in the incompleteness of the obtained results. In addition, threats might come from how the empirical study was carried out. To reduce the probability of having faults in our testing framework, it has been carefully tested. Nevertheless, it can be known that testing alone cannot prove the absence of defects.

With regard to *external validity* threats, this is an arduous issue, as generalized results depend on whether the models under test are representative of the targeted application domain and whether the faults taken into account are representative of real faults [1]. To cope with this problem, our

set of mutation operators at the design level for Simulink models is proposed by investigating the most common faults made by designers. Furthermore, experiments in this paper were run on several different Simulink models which contain the common blocks in the Simulink library used for many industrial models. Those models were manually chosen. However, to reduce the threat to validity, many further experiments should be carried out on large industrial models in the consequent studies.

6. CONCLUSION AND FUTURE WORK

Mutation coverage is an effective criterion that is able to support test data generation. This paper proposed the Improved Genetic Algorithm with the modifications in individual representation, the way of computing fitness function, the multi-parent crossover method, and the new mutation operator to generate test data for Simulink models based on mutation testing. Each individual is represented in the IGA by a test instead of a set of tests in the GA and SA, and the best individuals are stored for each genetic generation into the memory set. Three-parent crossover is also used instead of double points, and the new mutation operator is proposed to enhance the diversity of population. The fitness of each individual is computed based on the alive mutants in the current generation. The experimental results indicated that the IGA outperformed the GA, SA, and even AIS in terms of the number of killed mutants and the execution time.

Future work focuses on improving the fitness function towards a specific mutant in order to increase the effectiveness for the process of searching for test data. We are also going to carry out more experiments to determine the influence of the IGA configuration parameters on results as well as finding a way to automatically adjust these parameters in the search process.

REFERENCES

[1] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 742–762, 2010.

[2] B. Baudry, F. Fleurey, J. M. Jzquel, and Y. L. Traon, "Genes and bacteria for automatic test cases optimization in the .net environment," in *Proceedings of the 13th International Symposium on Software Reliability Engineering*, 2002, pp. 195–206.

[3] T. Blickle and L. Thiele, "A comparison of selection schemes used in evolutionary algorithms," *Evolutionary Computation*, vol. 4, no. 4, pp. 361–394, 1996.

[4] L. Bottaci, "A genetic algorithm fitness function for mutation testing," in *Proceedings of the Software Engineering using Metaheuristic inovative Algorithms workshop*, 2001, pp. 3–7.

[5] T. Y. Chen, H. Leung, and I. K. Mark, "Adaptive random testing," in *Proceedings of 9th Asian Computing Science Conference - Advances in Computer Science - ASIAN 2004: Higher-Level Decision Making*, 2004, pp. 320–329.

[6] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for practicing for programmer," *IEEE Computer*, vol. 11, pp. 34–41, 1978.

[7] R. A. Demillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transaction Software Engineering*, vol. 17, pp. 900–910, 1991.

[8] K. Derderian, M. Merayo, R. Hierons, and M. Nunez, "Aiding test case generation in temporally constrained state based systems using genetic algorithms," *Bio-Inspired Systems: Computational and Ambient Intelligence*, vol. 5517, pp. 327–334, 2009.

[9] K. Ghani, J. A. Clark, and Y. Zhan, "Comparing algorithms for search-based test data generation of matlab simulink model," in *Proceedings of 10th IEEE Congress on Evolutionary Computation*, 2007, pp. 2940–2947.

[10] Girgis, "Automatic test generation for data flow testing using a genetic algorithm," *Journal of Universal Computer Science*, vol. 11, no. 6, pp. 898–915, 2005.

[11] H. Haga and A. Suehiro, "Automatic test case generation based on genetic algorithm and mutation analysis," in *Proceedings of IEEE International Conference on Control System, Computing and Engineering*, 2012, pp. 119–123.

[12] L. Hanh, N. T. Binh, and K. T. Tung, "Applying the meta-heuristic algorithms for mutation-based test data generation for simulink models," in *Proceeding of the Fifth Symposium on Information and Communication Technology (SoICT2014)*, 2014, pp. 102–109.

[13] L. M. Hanh and N. T. Binh, "Automatic generation of mutants for simulink models," in *Proceedings of the 16th National Conference Selected Problems About IT And Telecommunication*, 2013, pp. 339–346.

[14] L. M. Hanh, N. T. Binh, and K. T. Tung, "Survey on mutation-based test data generation," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 5, no. 5, pp. 1164–1173, 2015.

[15] L. T. M. Hanh and N. T. Binh, "Mutation operators for simulink models," in *Proceedings of KSE 2012 - The fourth International Conference on Knowledge and Systems Engineering*, 2012, pp. 54–59.

[16] L. T. M. Hanh, N. T. Binh, and K. T. Tung, "A novel test data generation approach based upon mutation testing by using artificial immune system for simulink models," in *Proceedings of The Sixth International Conference on Knowledge and Systems Engineering*, 2014, pp. 169–181.

[17] L. T. M. Hanh, K. T. Tung, and N. T. Binh, "Mutation-based test data generation for simulink models using genetic algorithm and simulated annealing," *International Journal of Computer and Information Technology*, vol. 3, no. 4, pp. 763–771, 2014.

[18] J. H. Holland, *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[19] M. Last, S. Eyal, and A. Kandel, "Effective black-box testing with genetic algorithms," *Lecture notes in computer science, Springer*, vol. 3875, pp. 134–148, 2006.

[20] R. Lefticaru and F. Ipate, "Automatic state-based test generation using genetic algorithms," in *Proceedings of the Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2007, pp. 188–195.

[21] J. Louzada, C. G. Camilo-Junior, A. Vincenzi, and C. Rodrigues, "An elitist evolutionary algorithm for automatically generating test data," in *Proceedings of 2012 IEEE Congress on Evolutionary Computation (CEC)*, 2012, pp. 1–8.

[22] L. G. Madronal, J. J. D. Jimenez, and I. M. Bulo, "Mutation testing: Guideline and mutation operator classification," in *Proceedings of The Ninth International Multi-Conference on Computing in the Global Information Technology*, 2014, pp. 171–179.

[23] R. Malhotra, N. Singh, and Y. Singh, "Genetic algorithms: Concepts, design for optimization of process controllers," *Computer and Information Science*, vol. 4, no. 2, pp. 39–54, 2011.

[24] M. Masud, A. Nayak, M. Zaman, and N. Bansal, "Strategy for mutation testing using genetic algorithms," in *Proceedings of IEEE CCECE/CCGEI*, 2005, pp. 1049–1052.

[25] P. S. May, "Test data generation: Two evolutionary approaches to mutation testing," PhD Thesis, The University of Kent, 2007.

[26] P. McMinn, "Search-based software test data generation: A survey," *Software Testing Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[27] A. J. Offutt, R. Pargas, S. Fichter, and P. Khambekar, "Mutation testing of software using a mimd computer," in *Proceedings of International Conference on Parallel Processing (ICPP' 92)*, 1992, pp. 257–266.

[28] A. J. Offutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," Technical Report ISSE-TR-96-01, 1996.

[29] J. Oh, M. Harman, and S. Yoo, "Transition coverage testing for simulink/stateflow models using messy genetic algorithms," in *Proceedings of the 13th annual conference on Genetic and evolutionary computation (GECCO'11)*, 2011, pp. 1851–1858.

[30] C. Paduraru, M. C. Melemciuc, and A. Stefanescu, "A distributed implementation using apache spark of a genetic algorithm applied to test data generation," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2017, pp. 1857–1863.

[31] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE'10)*, 2010, pp. 121–130.

[32] X. Peng and L. Lu, "A new approach for session based test case generation by ga," in *Proceedings of IEEE 3rd International Conference on Communication Software and Networks (ICCSN)*, 2011, pp. 91–96.

[33] M. Roper, I. Maclean, and A. B. e. al., "Genetic algorithms and the automatic generation of test data," Department of Computer Science, University of Strathclyde, Technical Report RR/95/195[EFoCS-19-95], 1995.

[34] Y. Singh, "Automated expected output generation: Is this a problem that has been solved?" *SIGSOFT Software Engineering Notes*, vol. 40, no. 6, pp. 1–5, 2015.

[35] F. C. M. Souza, M. Papadakis, Y. L. Traon, and M. E. Delamaro, "Strong mutation-based test data generation using hill climbing," in *Proceedings of the 9th International Workshop on Search-Based Software Testing (SBST '16)*, 2016, pp. 45–54.

[36] Y. Suresh and S. K. Rath, "A genetic algorithm based approach for test data generation basis path testing," *The International Journal of Soft Computing and Software Engineering*, vol. 3, pp. 326–332, 2013.

[37] A. M. R. Vincenzi, T. Bachiega, D. G. d. Oliveira, S. R. S. d. Souza, and J. C. Maldonado, "The complementary aspect of automatically and manually generated test case sets," in *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, 2016, pp. 23–30.

[38] A. Windisch, "Search-based test data generation from stateflow statecharts," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, 2010, pp. 1349–1356.

[39] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering: An Introduction.* Kluwer Academic Publishers, 2000.

[40] A. H. Wright, *Genetic Algorithms for Real Parameter Optimization.* Morgan Kaufmann, 1991, pp. 205–218.

[41] S. Xanthakis, C. Ellis, and C. S. e. al., "Application of genetic algorithms to software testing," in *Proceedings of 5th International Conference on Software Engineering and its Applications*, 1992, pp. 625–636.

[42] X. Xu, Z. Zhu, and L. Jiao, "An adaptive fitness function based on branch hardness for search based testing," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2017, pp. 1335–1342.

[43] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2014, pp. 919–930.

[44] Y. Zhan, "A search-based framework for automatic test-set generation for matlab/simulink models," PhD Thesis, University of York, 2005.

[45] L. Zhang, T. Xie, and L. Z. e. al., "Test generation via dynamic symbolic execution for mutation testing," in *Proceedings of 2010 IEEE International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.

Received on March 13, 2017
Revised on September 28, 2017