A feedback-directed method of evolutionary test data generation for parallel programs

Dunwei Gong[1], Feng Pan[1], Tian Tian[2*], Su Yang[1], Fanlin Meng[3]

[1]*School of Information and Control Engineering, China University of Mining and Technology, Xuzhou 221116 Jiangsu, P. R. China*

[2]*School of Computer Science and Technology, Shandong Jianzhu University, Jinan 250101 Shandong, P. R. China*

[3] *Department of Mathematical Sciences, University of Essex, Colchester, CO4 3SQ, UK*

[*]*Corresponding author, email address: tian_tiantian@126.com.*

Abstract

Context: Genetic algorithms can be utilized for automatic test data generation. Test data are encoded as individuals which are evolved for a number of generations using genetic operators. Test data of a parallel program include not only the program input, but also the communication information between each pair of processes. Traditional genetic algorithms, however, do not make full use of information provided by a population's evolution, resulting in a low efficiency in generating test data.

Objective: This paper emphasizes the problem of test data generation for parallel programs, and presents a feedback-directed genetic algorithm for generating test data of path coverage.

Method: Information related to a schedule sequence is exploited to improve genetic operators. Specifically, a scheduling sequence is evaluated according to how well an individual covers the target path. The probability of the crossover and mutation points being located in the region is determined based on the evaluation result, which prevents a good schedule sequence from being destroyed. If crossover and mutation are performed in the scheduling sequence, the location of crossover and mutation points is further determined according to the relationship between nodes to be covered and the scheduling sequence. In this way, the population can be evolved in a narrowed search space.

Results: The proposed algorithm is applied to test 11 parallel programs. The experimental results show that, compared with the genetic algorithm without utilizing information during the population evolution, the proposed algorithm signif-

icantly reduces the number of generations and the time consumption.
Conclusion: The proposed algorithm can greatly improve the efficiency in evolutionary test data generation.

Keywords: information utilization, genetic algorithm, parallel program, path coverage, test data

---

NOMENCLATURE

Abbreviations

GA	Genetic Algorithm
WRNG	Wildcard Receiving Node Group
BGA	The basic GA
ESS-GA	The GA enhanced by Evaluating of Scheduling Sequences
RUS-GA	The GA enhanced by exploiting Relationship between Uncovered nodes and Scheduling sequences
ESS-RUS-GA	The algorithm combining ESS-GA with RUS-GA

Notations

$\boldsymbol{P}$	A parallel program
P^i	The $(i+1)$-th process in $\boldsymbol{P}$
n_j^i	The j-th node in P^i
p	A path in a parallel program
p^i	The path of process P^i and a sub-path in p
$\|p^i\|$	The length of p^i
p^*	A target path
$s(p^*, p)$	The similarity of the two pathes, p^* and p
r_j^i	A WRNG starting with n_j^i in P^i
X	The decision variable or an individual
D_X	The value domain of X
x_i	The i-th input variable
r_j	The scheduling sequence corresponding to the j-th WRNG
$F(X)$	The fitness of X
A	An archive that deposits the generated individuals
$\|A\|$	The size of A
m	The number of scheduling sequences in A
s_i	The i-th scheduling sequence in A

$\{s_i\}$	The set of individuals with s_i in A
$F(\{s_i\})$	The evaluation of s_i
R	The set of all the WRNGs in $\boldsymbol{P}$
n_i	The i-th uncovered node in the target path
c	The number of uncovered nodes in the target path
R_i	The set of all the WRNGs that influence n_i
R_i^1	The set of WRNGs in R that are located at the same process as n_i
R_i^2	The set of WRNGs in R that are located at different processes with n_i
g	The number of generations
t	The time consumption
R_g	The reduction rate in the number of generations
R_t	The reduction rate in the time consumption

1. Introduction

Software testing is an important method to guarantee software quality and focuses on detecting errors in a software product [1]. Only when test data touch an element of a program in execution, bugs associated with the element could be potentially detected. In other words, test data that cover the element of a program are necessary for detecting its bugs. Therefore, generating test data for a predefined coverage criterion plays a very important role in testing a software [2]. This study emphasizes the problem of test data generation and proposes an enhanced genetic algorithm based on feedback information during the evolution to tackle this problem.

A parallel program, containing two or more processes that execute in parallel [3], can conduct large-scale parallel computing. Although there are other ways to design a parallel program, extending traditional programming languages by using message-passing environments is the most commonly used method [4]. In addition, message-passing interface(MPI) has become one of the most notable and important parallel programming environments [5]. In view of this, we focus on the problem of test data generation for message-passing parallel programs in this paper.

Path coverage has a strong capability of detecting faults [6], and is widely used in software testing [7]. The specific implementation of generating test data for path coverage is as follows: for a target path of a program given by a tester, seeking test data in the input domain of the program against which the covered path is exactly the target one [8, 9]. To generate test data for path coverage automatically and efficiently, an optimization problem can be formulated and solved

by optimization methods, especially by search-based methods. A lot of attempts using search-based methods to solve the problem have been reported in the literature [10], such as genetic algorithms(GAs) [11], simulated annealing [12], and particle swarm optimization [13]. In addition, recent emerging meta-heuristics techniques [14], such as firefly [15–17], cuckoo search [18], bee algorithm [19] and bat algorithm [20], can also be applied in search-based software testing. Among these methods, GAs are very popular and have advantages in solving problems with large-scale search spaces and nonlinear objectives [21, 22].

MPI library provides various functions to implement either point-to-point or collective communication, where point-to-point communication means sending a piece of message from one process to another, whereas collective communication signifies a one-to-many or many-to-many massage passing pattern [23]. With respect to point-to-point communication, a receiving statement, which calls a MPI primitive, can receive a piece of message from any source due to the wildcards, MPI _ANY_SOURCE and MPI_ANY_TAG [24]. The receiving sequence of these statements forms a scheduling sequence, resulting in the non-deterministic program execution, which increases the difficulty of test data generation [25].

Taking the feature of the non-deterministic execution of a parallel program into consideration, this paper focuses on an enhanced GA based on feedback information during the evolution to generate test data for path coverage. The proposed GA takes full advantage of information provided by a population's evolution to evaluate scheduling sequences to guide the subsequent evolution, and analyzes the relationship between uncovered nodes and scheduling sequences to narrow down the search space. More specifically, a scheduling sequence is firstly evaluated according to the relationship between the execution path of an evolutionary individual and the target path. Following that, the probability of the crossover and mutation points being located in any particular region is determined based on the above evaluation. If the crossover and mutation points are located in the scheduling sequence, the subsequent search space can be further narrowed down according to the relationship between uncovered nodes and scheduling sequences.

The remainder of this paper is organized as follows. Section 2 reviews the related work. The preliminary knowledge of message-passing parallel programs is introduced in Section 3. Section 4 details the proposed approach, which includes methods of evaluating a scheduling sequence, determining the probability of the crossover and mutation points being located in the region, seeking the relationship between uncovered nodes and scheduling sequences, and locating the crossover and mutation points in the scheduling sequence. Experimental results and analysis are provided in Section 5. Finally, Section 6 concludes the paper.

2. Related Work

If a program includes multiple execution flows that progress simultaneously and interact with each other, it is called a concurrent program [26]. Concurrent program testing has received increasing attention from the community of software testing [26, 27]. The following reviews the related work from the aspects of multi-thread program testing and multi-process program testing. Here, a multi-thread program means a program that adopts the mechanism of shared-memory for interleaving execution flows, while a multi-progress program employs message passing to perform the interaction between execution flows within distributed-memory environments.

Nistor et al. [28] proposed a random-based approach to producing multi-threaded test data to trigger concurrency bugs. Terragni et al. [29] proposed a coverage-based approach which develops concurrency-related coverage requirements and generates method call sequences that achieve such requirements. Given that these methods are subject to special kinds of concurrency bugs and demand an expensive computation, Choudhary et al. [30] presented a coverage-guided approach which can detect arbitrary concurrency bugs with an inexpensive computation.

Symbolic execution is a promising technique for software testing and replaces a concrete program input with the symbolic one to run a program [31]. Guo et al. [32] used the method of symbolic execution to explore intra-thread paths and inter-thread interleaving of a multi-thread program. [33] further extended the above work to regression testing of multi-thread programs to explore interleaving affected by changed codes. In addition, Zhang et al. [34] emphasized definition-use data flow instead of interleaving. Khanna et al. [35] combined the symbolic execution and dynamic verification to analyze MPI programs.

Reachability testing is another important approach to testing multi-thread programs. It is based on prefix-based testing, which allows test data to run deterministically up to a desired program state, and thereafter run to proceed non-deterministically [36]. Carver et al. [37] proposed a distributed reachability testing algorithm. Qi et al. [38] presented another reachability testing method, which integrates the variable strength strategy into a testing framework to balance the effectiveness and the efficiency. [39] also claimed a parallel approach for reachability testing.

Based on the above, studies on multi-thread program testing mainly focus on method sequence generation and thread interleaving exploration. Different from them, detecting deadlock and resource competition are the focus of multi-process parallel program testing. Along this line, Vakkalanka et al. [40] proposed

a model-checking tool which executes all the processes of a parallel program by an interleaving scheduler. Leungwattanakit et al. [41] proposed a cache-based model checking method for distributed parallel programs. Vetter et al. [42] detected programming errors in parallel programs by monitoring MPI operations. Park et al. [43] checked the communication concurrency between processes for detecting and reporting race conditions. Krammer et al. [44] checked the usage of Application Program Interface of MPI at run time. Ayub et al. [45] implemented model checking for MPI Java programs and modeled processes as threads. In addition, Gong et al. [46] presented an approach to reducing process interleaving when generating test data for statement coverage.

Due to the high adaptability, parallelism and global search ability [47, 48], GAs have been the most widely used search-based methods in the literature for generating test data of parallel programs. Tian et al. [49] proposed a co-evolutionary GA to generate test data for path coverage of message-passing parallel programs. However, the work does not take the non-determinism of parallel programs into consideration. In other words, the method in [49] focuses on a kind of parallel programs with the deterministic execution, whereas this study emphasizes non-deterministic parallel programs. In addition, Cao et al. [50] generated test data under the strong mutation testing criterion by combining the symbol execution with evolutionary algorithms. Ghiduk et al. employed genetic algorithms to generate test data that kill concurrency mutants [51]. Anbunathan et al. utilized genetic algorithms when generating test data from activity diagrams with concurrent behaviors [52].

Feedback-directed mechanism has been incorporated into software testing techniques. Along this line, Pacheco et al. [53] [54] proposed a feedback-directed random test data generation method to avoid redundant or illegal method sequences. Tan et al. [55] employed feedback-directed GA to optimize software configurations. When evolutionarily generating test data, Dang et al. [56] also dynamically reduced the search domain based on the optimal solution of each population. In addition, Luo et al. [57] proposed a feedback-directed method for performance testing.

Table 1 summarizes the software testing techniques proposed for multi-thread and multi-process programs in terms of bugs-oriented test data generation, symbolic execution, reachability testing, deadlock detection and resource competition, GA-based test data generation, and feedback-directed mechanism in software testing. In particular, for GA-based test data generation, Table 2 analyses the current research gaps by specifying the issues identified from related works, and presents how the proposed method could fill the gap.

Table 1: Summary of test data generation techniques proposed in the context of multi-thread and multi-process program testing.

Techniques of test data generation	Sources of literature
Bugs-oriented test data generation	[28–30]
Symbolic execution for parallel programs	[32–35]
Reachability testing	[36–39]
Deadlock detection and resource competition	[40, 41, 43–46]
GA-based test data generation for parallel programs	[49–52]
Feedback-directed mechanism in software testing	[53–57]

Table 2: Issues identified from related works and their relation to the proposed solution of this study

Open questions in literatures	Solutions in the proposed method
(1) The feedback-directed mechanism is not applied in GA-based test data generation for unit testing [49–52].	Presenting a GA-based test data generation method incorporating the information feedback.
(2) The test data generation method is for the parallel programs with a fixed schedule sequence and no non-determinism [49].	Confronting the impacts of different schedule sequences on the execution trace of parallel programs
(3) The information during the evolution is not utilized to improve the efficiency of test data generation [49, 50].	Utilizing the feedback information related to the schedule sequence during the evolution.

This paper takes the characteristics of message-passing parallel programs into consideration, and proposes a feedback-directed GA that focuses on the utilization of feedback information during the evolution, so as to enhance the efficiency of test data generation. Before we illustrate the proposed approach, the background related to message-passing parallel programs is first introduced as follows.

3. Foundation of evolutionary test data generation for Parallel Programs

3.1. Notions for Parallel Programs

A **parallel program** contains $num(num > 1)$ processes executed in parallel and can be represented as $\boldsymbol{P} = \{P^0, P^1, \ldots, P^{num-1}\}$. Fig. 1(a) shows an example of a parallel program, which includes four processes, that is, $\boldsymbol{P} = \{P^0, P^1, P^2, P^3\}$. For the purpose of clarity, the functions, send and recv, are used for placing MPI primitives to interpret sending and receiving a piece of message. For example, send(a1,1) in P^0 means that P^0 sends the value of $a1$ to P^1 while recv(a1,0) in P^0 implies that P^1 receives a value from P^0 and stores it by $a1$. Specifically, recv(x1,ANY) in P^0 refers to that P^0 receives a piece message from any process and stores it by $x1$.

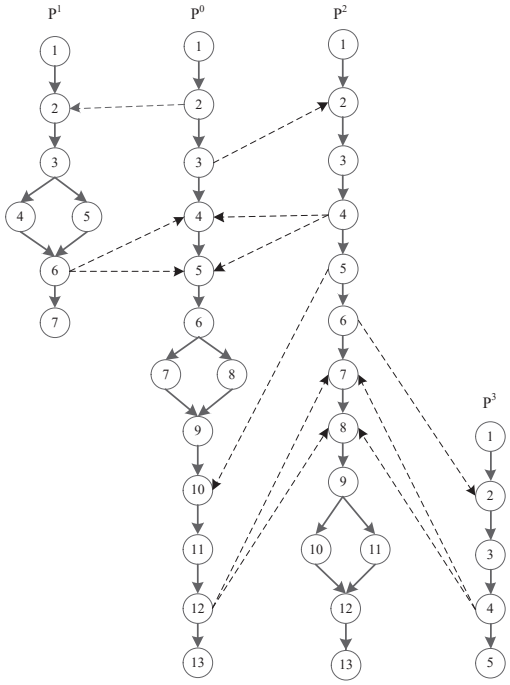
```
P0
1 initialize( );
  int a1,a2,b,x1,x2,y1,z;
2 send(a1,1);
3 send(a2,2);
4 recv(x1,ANY);
5 recv(x2,ANY);
6 if(x1>x2)
7    z=x1-x2;
else
8    z=x2-x1;
9 printf("z=",z);
10 recv(b,2);
11 y1=(b-5)*z;
12 send(y1,2);
13 finalize( );
```

```
P1
1 initialize( );
  int a1, x1;
2 recv(a1,0);
3 if(a1>=0&&a1<=20)
4        x1=a1*2;
else
5        x1=a1*3;
6 send(x1,0);
7 finalize( );
```

```
P3
1 initialize( );
  int b, y2;
2 recv(b,2);
3 y2=b-15;
4 send(y2,2);
5 finalize( );
```

```
P2
1 initialize( );
int a2,x2,b,y1,y2,t;
2 recv(a2,0);
3 x2=a2/2;
4 send(x2,0);
5 send(b,0);
6 send(b,3);
7 recv(y1,ANY);
8 recv(y2,ANY);
9 if(y1*y2>=0)
10      t=y1;
   else
11     t=y2;
12 printf("t=",t);
```

(a) Example program



(b) Flow chart

Figure 1: An example of the message-passing parallel program and its flow chart.

Let us define a **basic block** as a node in which either all its statements are executed or none of them is executed. For the process, P^i, in ***P***, its j-th node is denoted as n_j^i. In Fig. 1(a), n_1^0 implies the first basic block of P^0. In the example program, P^0 sends $a1$ and $a2$ to P^1 and P^2, respectively. P^1 receives the value sent from P^0 for making some calculations and returns the result to P^0. P^2 also takes the value from P^0 and sends the results to P^0. P^0 receives message from P^1 and P^2, gets the value of z, receives b from P^1, and finally sends $y1$ to P^2. P^2 sends b to P^0 and P^3, and receives message from P^1 and P^3. Accordingly, some operations are performed. P^3 receives b from P^2, and sends the calculation result to P^2. The four processes execute in parallel and block when encountering sending and receiving operations, until the operation is finished.

For any two nodes, n_k^i and n_l^i in P^i, if n_k^i is executed immediately after n_l^i against a certain input, there will be a control edge between the above two nodes, denoted as $< n_k^i, n_l^i >$. Assume n_k^i is a node that sends a piece of message (sending node, for short) and n_l^j is a receiving node that matches with the sending node in P^j. There will be a communication edge, $< n_k^i, n_l^j >$. For example, in the flow chart of the example program, Fig. 1(b), $< n_2^0, n_2^1 >$ is a communication edge between P^0 and P^1, and $< n_6^0, n_7^0 >$ is a control edge in P^0.

3.2. Path coverage problem of parallel programs

If ***P*** is executed against a program input, a sequence of nodes that the input traverses constitute a path, denoted as $p = p^0 p^1 \ldots\ldots p^{num-1}$. Here, $p^i, i = 0, 1, ..., num-1$, is the sub-path of process i. The number of nodes contained in p^i is referred to as the length of the sub-path, denoted as $|p^i|$. For two paths, $p^* = p^{0^*} p^{1^*} ... p^{num-1^*}$ and $p = p^0 p^1 ... p^{num-1}$, their similarity is calculated by [49]:

$$s(p^*, p) = \frac{1}{num} \sum_{i=0}^{num-1} \frac{|p^{i^*} \cap p^i|}{\max\{|p^{i^*}|, |p^i|\}} \tag{1}$$

where $|p^{i^*} \cap p^i|$ is the number of consecutively identical nodes when comparing the sub-paths of process i in these two paths, i.e., p^{i^*} and p^i, from the first node. For two sub-paths

$$p^{0^*} = n_1^0 n_2^0 n_3^0 n_4^0 n_5^0 n_6^0 n_7^0 n_9^0 n_{10}^0 n_{11}^0 n_{12}^0 n_{13}^0$$

and

$$p^0 = n_1^0 n_2^0 n_3^0 n_4^0 n_5^0 n_6^0 n_8^0 n_9^0 n_{10}^0 n_{11}^0 n_{12}^0 n_{13}^0$$

of the process, P^0, in Fig. 1, both $|p^{0^*}|$ and $|p^0|$ are 12, and $|p^{0^*} \cap p^0|$ is 6.

In ***P***, there may exist some wildcard receiving statements which do not specify their message sources. When more than one sending statements match a wildcard receiving statement, different executions of a parallel program under the same input may experience different traces, which is called **non-determinism**.

More specifically, assume that $n^i_{j1}, ..., n^i_{jm}$ are m wildcard receiving statements in P^i with the same message envelope, and each of them matches the same sending statements, then $n^i_{j1}, ..., n^i_{jm}$ constitute a **Wildcard Receiving Node Group** (**WRNG**, for short), denoted as r^i_j. Its receiving sequence is composed of communication edges whose receiving nodes belong to the WRNG. Consequently, a **scheduling sequence** consists of receiving sequences of all WRNGs. A program may traverse different paths in different runs with the same input due to different scheduling sequences.

For instance, in the program shown in Fig. 1, n^0_4 and n^0_5 form a WRNG, denoted as r^0_4, while n^2_7 and n^2_8 form another WRNG, r^2_7. $(< n^1_4, n^0_4 >, < n^2_4, n^0_5 >)$ is a receiving sequence of r^0_4 in addition to $(< n^2_4, n^0_4 >, < n^1_4, n^0_5 >)$. Similarly, $(< n^0_{12}, n^2_7 >, < n^3_4, n^2_8 >)$ and $(< n^3_4, n^2_7 >, < n^0_{12}, n^2_8 >)$ are two receiving sequences corresponding to r^2_7. Correspondingly, $(< n^2_4, n^0_4 >, < n^1_4, n^0_5 >, < n^0_{12}, n^2_7 >, < n^3_4, n^2_8 >)$ is a scheduling sequence. The order of receiving messages in r^0_4 and r^2_7 has an influence on the execution trace of the example program.

3.3. Notions for evolutionary test data generation

For a target path, p^*, when the decision variable, which is denoted as X in the range of D_X executes ***P***, the traversed path is denoted as $p(X)$. The problem of generating test data for path coverage can be formulated as:

$$\begin{aligned} &\max s(p^*, p(X)) \\ &\text{s.t. } X \in D_X \end{aligned} \tag{2}$$

Since the closeness between the path covered by an individual, X, and the target path can be reflected by $s(p^*, p(X))$, we represent the fitness of the individual as

$$F(X) = s(p^*, p(X)) \tag{3}$$

For two individuals, $X_1 = (x_{11}, x_{12}, ..., x_{1i}, ..., x_{1n_s}, r_{11}, r_{12}, ..., r_{1n_r})$ and $X_2 = (x_{21}, x_{22}, ..., x_{2i}, ..., x_{2n_s}, r_{21}, r_{22}, ..., r_{2n_r})$, we perform the single-point crossover operation on them, and the crossover point is located in the i-th component of the program input, then two new individuals, denoted as $X_1^{'}$ and $X_2^{'}$, will be generated. Further, we conduct the single-point mutation on X_1, the mutation point is also located in the i-th component of the program input and the new allele at

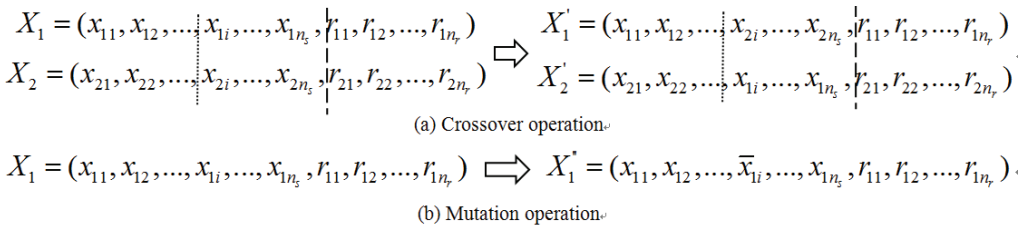


Figure 2: The process of the crossover and mutation operations.

this locus is $\bar{x}_{1i}$, then the individual after mutation, denoted as X_1'', will also be produced. The process of the crossover and mutation operations is depicted as Fig. 2.

4. The Proposed Approach

This section presents a new evolutionary optimization method with the purpose of improving the efficiency in generating test data, which will be described as follows. The probability of the crossover and mutation points being located in the region is determined based on the evaluation of scheduling sequences. In the case that the crossover and mutation points are located in the scheduling sequence, the range of the crossover and mutation points can be further narrowed down according to the relationship between uncovered nodes and scheduling sequences.

4.1. The Evaluation of a Scheduling Sequence

Note that the decision variable of the problem that generates test data for path coverage formulated in (2) contains the program input and the scheduling sequence. As a result, an individual obtained by encoding methods also consists of these two parts. In the following, the method of evaluating a scheduling sequence will be given.

Starting from an initial population, traditional evolutionary optimization is employed for a number of generations, and all the generated individuals are put into an archive, denoted as A, with its size being $|A|$, until $|A|$ is equal to a threshold value set in advance. The scheduling sequences contained in A are retrieved. Based on these scheduling sequences, individuals in A are divided into a number of classes, with each representing a set of individuals with the same scheduling sequence. Assume that m is the number of scheduling sequences and $s_1, s_2, \cdots,$

and s_m are the scheduling sequences in A. The i-th class is presented as $\{s_i\}$, which contains $X_{i1}, X_{i2}, \cdots, X_{i|\{s_i\}|}$ with $|\{s_i\}|$ representing the number of individuals in $\{s_i\}$. Based on formula (3), the fitness of X_{ij}, denoted as $F(X_{ij})$, $j = 1, 2, \cdots, |\{s_i\}|$, can be calculated. Finally, s_i can be evaluated as follows:

$$F(\{s_i\}) = \frac{1}{|\{s_i\}|} \sum_{j=1}^{|\{s_i\}|} F(X_{ij}) \tag{4}$$

4.2. The Probability of the Crossover and Mutation Points Located in the Scheduling Sequence

To determine the probability, the influence of a scheduling sequence on the target path coverage is firstly analyzed. It is evident that different scheduling sequences have their own difficulties in covering the target path, which can be observed from the fact that a number of program inputs are able to cover the target path under one scheduling sequence, whereas only a few of them can cover it under another scheduling sequence. Sometimes, none of the program inputs can cover the target path under a specific scheduling sequence.

In the program shown in Fig. 1(a), $(< n_4^1, n_4^0 >, < n_4^2, n_5^0 >, < n_{12}^0, n_7^2 >, < n_4^3, n_8^2 >)$ and $(< n_4^2, n_4^0 >, < n_4^1, n_5^0 >, < n_{12}^0, n_7^2 >, < n_4^3, n_8^2 >)$ are two scheduling sequences associated with r_4^0 and r_7^2, respectively. The ranges of the program input, a_1, a_2, and b, are $[0, 20]$. To cover the following target path,

$$\begin{gathered} p^* = n_1^0 n_2^0 n_3^0 n_4^0 n_5^0 n_6^0 n_7^0 n_9^0 n_{10}^0 n_{11}^0 n_{12}^0 n_{13}^0 \\ n_1^1 n_2^1 n_3^1 n_4^1 n_5^1 \\ n_1^2 n_2^2 n_3^2 n_4^2 n_5^2 n_6^2 n_7^2 n_8^2 n_9^2 n_{11}^2 n_{12}^2 n_{13}^2 \\ n_1^3 n_2^3 n_3^3 n_4^3 n_5^3 \end{gathered}$$

under the first scheduling sequence, the program input is given as $\{(a_1, a_2, b) | \frac{a_2}{4} < a_1 \leq 20, 0 \leq a_2 \leq 20, 5 < b < 15\}$ whereas, for another scheduling sequence, the program input becomes $\{(a_1, a_2, b) | 0 \leq a_1 < \frac{a_2}{4}, 0 \leq a_2 \leq 20, 5 < b < 15\}$. Given the fact that under the first scheduling sequence, the program input domain is much lager than that under the second scheduling sequence, generating test data to cover the target path is much easier under the first scheduling sequence.

For a scheduling sequence under which the target path is easy to cover, the destruction to such a scheduling sequence should be as little as possible when conducting the crossover and mutation operations. Therefore, the crossover and mutation points should have a low probability of being located in such scheduling sequence. Otherwise, they should have a high probability to be located in it.

From formulas (1), (3), and (4), $F(\{s_i\})$ is in the range of $[0, 1]$. If a scheduling sequence is assumed as good based on $F(\{s_i\})$, the crossover and mutation operations should not impact the chromosome encoding of the sequence much. In other words, a good scheduling sequence always means a low probability of the crossover and mutation points being located in the sequence. Accordingly, the probability of applying the crossover and mutation operations in the scheduling sequence is given as $1 - F(\{s_i\})$ while the probability of conducting crossover and mutation in the program input is assigned as $F(\{s_i\})$.

During the population evolution, A is continuously updated along with the generation of new individuals, and the probabilities are thus updated based on information provided by elements in A until the termination criteria of the algorithm are met.

4.3. The Location of the Crossover and Mutation Points in a Scheduling Sequence

(1) WRNGs That Affect Uncovered Nodes

If the operations are conducted on the scheduling sequence, we need to further determine the specific positions of crossover and mutation points. To this end, we analyze the influence of WRNGs on uncovered nodes. The reason lies in that the path covered by an individual consists of a series of nodes, in which some are the same as those of the target path, whereas the others are not. Such different nodes will make the path covered by the individual different from the target path. For an uncovered node in the target path, it is affected by some or all the WRNGs. As a result, it is possible to cover this node by changing the order of a WRNG, i.e., changing the scheduling sequence. Since the order of the scheduling sequence can be changed by crossover and mutation operations, determining the specific positions of crossover and mutation points becomes very important to conduct such operations.

Although a parallel program generally has multiple WRNGs, not all of them have an influence on an uncovered node. Considering that different WRNGs have different positions and thus different influences on an uncovered node, we can classify these WRNGs based on their positions.

Denote the set formed by all the WRNGs in a program as R , and the i-th uncovered node of the target path as $n_i, i = 1, 2, \cdots, c$, where c is the number of uncovered nodes. The set of all the WRNGs that have an influence on n_i is denoted as R_i. Initially, $R_i = \emptyset$. The elements in R can be divided into the following two subsets based on whether they are located at the same process as n_i. One is formed by WRNGs that are located at the same process as n_i, denoted as R_i^1, and the other consists of WRNGs that are located at different processes

from n_i, denoted as R_i^2. If $R_i^1 \neq \phi$, WRNGs that have front positions than n_i may have an influence on n_i, and will be put into R_i.

As for R_i^2, the influence of its WRNGs on n_i resulted from communications among processes is investigated. To fulfill this task, the receiving nodes located before n_i are considered, and all their matched sending nodes form a subset. For each sending node in the subset, the WRNGs in R_i^2 located before the sending node are sought. If at least one variable in a WRNG affects the variable(s) sent by the sending node, the WRNG will be put into R_i.

Using the above method, we can get the corresponding WRNGs for each uncovered node. As a result, the WRNGs that affect all the uncovered nodes can now be obtained, which form a set, denoted as $\overset{c}{\underset{i=1}{\cup}} R_i$.

For the program shown in Fig. 1, $R = \{r_4^0, r_7^2\}$. Suppose that the program input is $a_1 = 10, a_2 = 10, b = 3$, and the scheduling sequence is $(< n_4^2, n_4^0 > , < n_4^1, n_5^0 >, < n_{12}^0, n_7^2 >, < n_4^3, n_8^2 >)$. Then the executed path will have two uncovered nodes (i.e., n_7^0 and n_{11}^2) compared with p^*. For convenient illustration, the two nodes are denoted as n_1 and n_2. For n_2, initially $R_2 = \phi$, $R_2^1 = \{r_7^2\}$ and $R_2^2 = \{r_4^0\}$. Since r_7^2 in R_2^1 is located before n_2, it may have an influence on n_2. Therefore, r_7^2 is put into R_2. After this, we have $R_2 = \{r_7^2\}$.

The receiving nodes, n_7^2 and n_8^2, are located before n_2, and their matched sending nodes are n_{12}^0 and n_4^3, respectively. Since r_4^0 in R_2^2 is located before n_{12}^0, and a variable , (x_1) , in r_4^0 affects variable y_1 sent by n_{12}^0, r_4^0 is therefore put into R_2. Then we have $R_2 = \{r_4^0, r_7^2\}$. Note that for the sending node, n_4^3, there are no WRNGs in R_2^2 that are located before n_4^3. As a result, the set of WRNGs affecting the uncovered node, n_2, is finally given as $R_2 = \{r_4^0, r_7^2\}$.

Similarly, for the uncovered node, n_1, $R_1 = \{r_4^0\}$. WRNGs that affect the above two uncovered nodes, n_1 and n_2, form the set represented as $R_1 \cup R_2 = \{r_4^0, r_7^2\}$.

(2) The Crossover and Mutation Points in a Scheduling Sequence

When the crossover and mutation operations are conducted on the scheduling sequence to generate individuals for covering the target path, we aim to change the encoding order of WRNGs that affect uncovered nodes and to retain the encoding order of the other wildcard receiving nodes unchanged. As a result, the crossover and mutation points should be chosen at the part of the encoding corresponding to the WRNGs that affect uncovered nodes.

Suppose that there are two individuals, X_1 and X_2. Their covered paths can be obtained after decoding and executing the program under test. By comparing each of the two paths with the target one, the uncovered nodes of the target path can

also be achieved. Furthermore, WRNGs that affect uncovered nodes in the above two paths are denoted as $R(X_1)$ and $R(X_2)$, respectively, and can be obtained by using the method proposed in the previous subsection.

Following that, the crossover point is randomly selected among the encoding that corresponds to the WRNGs in $R(X_1)$ and $R(X_2)$. The offspring individual of X_1 consists of the following two parts: one owns the genes before the crossover point, which has the same alleles as those of X_1; the other has the genes after the crossover point. For the latter, if the WRNGs corresponding to the allele of a particular locus belong to $R(X_1)$, the allele of this locus will be swapped with that in X_2; otherwise, it will remain unchanged. The offspring individual of X_2 can also be generated similarly by the above approach.

Processes where the sending nodes are located can be employed to represent the receiving sequence of a WRNG. As a result, the receiving sequence, $(< n_4^1, n_4^0 >, < n_4^2, n_5^0 >)$, of r_4^0 can be denoted as (P^1, P^2). Correspondingly, assume that two test data for the program shown in Fig. 1 are $X_1 = (4, 18, 8, P^1, P^2, P^3, P^0)$ and $X_2 = (10, 10, 3, P^2, P^1, P^0, P^3)$, respectively, the individuals corresponding to them are obtained by encoding the program input with binary encoding and the scheduling sequence with integer encoding, which are given as follows: $X_1 = (0010010010010001230)$ and $X_2 = (0101001010000112103)$.

It is worth noting that an offspring may have an illegal scheduling sequence after conducting the crossover and mutation operations. The following method can be adopted to repair it. Starting from the first locus, we investigate whether it has the same encoding as each of the others. If yes, the encoding at this locus will be changed into a different one; otherwise, it will remain unchanged. Following the rule, the encoding at each locus before the crossover point is checked until the scheduling sequence becomes legal.

The sets of WRNGs that affect all the uncovered nodes in X_1 and X_2 are achieved via the method proposed in the previous subsection, and are given as follows: $R(X_1) = \{r_4^0\}$ and $R(X_2) = \{r_4^0, r_7^2\}$. The crossover point is randomly selected between the locus corresponding to the WRNGs in $R(X_1)$ and $R(X_2)$. That is, the locus is located between the 16th and the 19th loci. Suppose the crossover point is selected at the 17th locus, as r_4^0 is in $R(X_1)$, its corresponding encoding will be changed. In contrast, r_7^2 is not in $R(X_1)$, therefore its corresponding encoding will remain unchanged. By further performing the repair strategy, the offspring individual of X_1 becomes $X_1{}' = (0010010010010002130)$. Similarly, the offspring individual of X_2 becomes $X_2{}' = (0101001010000111230)$.

As for the mutation operation, if X_1 is selected for mutation, the mutation point should be chosen at the locus corresponding to the WRNG in $R(X_1)$. Fol-

lowing that, a traditional mutation operation is employed to generate the offspring individual. Similarly, the offspring individual can be further repaired if necessary.

4.4. Pseudo-code of the Proposed Method

To sum up, the pseudo-code of the method proposed in this paper is provided in Algorithm 1. In each generation, the fitness of each individual is calculated in Line 4. If test data that cover the target path are generated, the iteration will be terminated in Line 6. Otherwise, the genetic operation are applied on the next evolution in Lines 8-20. Lines 11-16 evaluate the performance of a scheduling sequence and further select the location of the crossover and mutation operations based on subsection 4.2. In Lines 17-18, if the location is in the scheduling sequence, the WRNGs that affect uncovered nodes will be determined and the crossover and mutation operations will be performed according to subsection 4.3. Overall, the proposed method improves the techniques of evolutionary test data generation for parallel programs from the following two aspects.

(1) The probability of which region the crossover and mutation points are located is determined based on the **E**valuation of **S**cheduling **S**equences, which corresponds to the first contribution of this study. The GA that employs the above method is called ESS-GA.

(2) The indexes of the decision variables where the crossover and mutation operations can be applied is reduced according to the **R**elationship between **U**ncovered nodes and **S**cheduling sequences. The corresponding GA is referred to as RUS-GA and maps the second contribution of this study.

Finally, the proposed method that integrates the above two improvements is described in Algorithm 1 and called ESS-RUS-GA. Correspondingly, the critical parts for ESS-GA and RUS-GA are highlighted in Algorithm 1.

5. Experiments

In this section, the proposed method is applied to test eleven benchmark programs and evaluated through a series of experiments. The questions to answer in the experiments are firstly raised. Secondly, the benchmark programs are briefly illustrated. Following that, the experimental environment and process are introduced. Finally, the experimental results are provided and analyzed and the main threats to validity are illustrated.

Algorithm 1 ESS-RUS-GA

Require: a program, ***P***, and a target path, p^*.

$A \leftarrow \emptyset$, $p_{s_i} \leftarrow 0.5$. /* p_{s_i} is the probability of the crossover and mutation points located in the scheduling sequence, s_i, of an individual.*/

Generate an initial population, $pop(1)$.

for k from 1 to Count, step 1 **do** /* Count is the maximal number of generations.*/

    Calculate the fitness value of each individual in $pop(k)$.

    if a test datum that covers p^* is generated **then**

        Print the test datum.

        Break.

    else

        Conduct selection in $pop(k)$.

- -$ESS-GA$-

        $A \leftarrow pop(k)$.

        if the threshold value of A is achieved **then**

            Calculate $F(\{s_i\})$.

            $p_{s_i} \leftarrow 1 - F(\{s_i\})$.

            $A \leftarrow \emptyset$.

        end if

        Conduct crossover and mutation based on the updated p_{s_i}.

- -

- -$RUS-GA$-

        if crossover(mutation) is performed at the scheduling sequence **then**

            Locate the point of crossover(mutation) operations.

- -

        end if

    end if

end for

5.1. The Questions to Answer

In order to evaluate the proposed method, ESS-GA should be evaluated to verify that the efficiency of evolutionary generation of test data can be improved by exploiting feedback information related to the performance of scheduling sequences in evolution. Similarly, RUS-GA is implemented for evaluating the utilization of feedback information of WRNGs that impact on uncovered nodes. Finally, the combination of the two techniques, ESS-RUS-GA, should be investigated for verifying how it contributes the efficiency of test data generation. Therefore, taking basic GA (abbreviated as BGA) as a baseline method, the following three questions are raised:

Q1: Can the efficiency in generating test data for path coverage be improved by using ESS-GA?

Q2: Can the efficiency in generating test data for path coverage be improved by using RUS-GA?

Q3: Can the efficiency in generating test data for path coverage be further improved by using ESS-RUS-GA? In addition, the following question is also be presented to investigate the difference between the proposed method and other test data generation techniques.

Q4: What is the difference between the proposed method and model checking as well as symbolic execution?

5.2. The Programs Under Test

Eleven benchmark programs are selected. Firstly, *Max_triangle* seeks the largest three from four numbers and judges whether they can constitute a triangle. For *Min*, it seeks the minimum one of a series of numbers. Regarding *Index* and *ASCII*, they are parallelization of common functions which retrieve the character information and judge the type of elements in a character string, respectively. With respect to *Including*, it judges the relationship in position between a point and a polygon. For *Matrix*, it multiplies two matrixes and seeks for two maximum values in the resulting values. In addition, *Convex_quadrilateral* and *Creator_consumer* judge whether four angles can constitute a convex quadrilateral, and simulate the process of production and consumption, respectively. They are often utilized for coverage testing of MPI programs, which are selected from [46]. Secondly, *Integrate_mw* calculates the integral of a trigonometric function, which is selected from the FEVS benchmark parallel programs [58]. *Search_function* is composed of Search and Function, where Search seeks a series of numbers in an array, and Function stems from mpi_mm and mpi_wave in [59], with its function

of matrices multiplication and solving a concurrent wave equation. Finally, *Kfray* from [60] is a ray tracing program that can create realistic images.

Concretely, *Search_function* has the largest number of processes, 70, whereas *Matrix* has the smallest number of processes, 4. These programs have in general various numbers of input variables, among which *Min* has the largest number of input variables, 125, whereas *Including*, *Creator_consumer*, *Integrate_mw*, and *Search_function* have only two input variables. Similarly, they are diverse in the number of wildcard receiving statements with the maximum of 130 and the minimum of 3. The above aspects emphasize that there are various difficulties in testing these programs. As a result, they are representative as the programs under test, which is beneficial to demonstrating the applicability and scalability of the proposed method.

Table 3: Basic information of programs under test.

Programs under test	# of processes	# of program input	# of wildcard receiving statements	Lines of code	# of target paths
Max_triangle	12	4	9	174	5
Including	10	2	8	191	7
Matrix	4	16	3	117	8
Index	7	15	5	157	7
Min	6	125	5	145	6
Convex_quadrilateral	9	4	7	130	9
ASCII	8	10	6	124	6
Creator_consumer	14	2	9	169	7
Integrate_mw	27	2	12	231	12
Search_function	70	2	130	1693	26
Kfray	8	20	116	12728(4129)	66

5.3. Experimental Environment and Process

Regarding the GA for generating test data to cover the target path, roulette wheel selection, single-point crossover and single-point mutation operators are adopted, and the probabilities of the crossover and mutation operators are set to 0.9 and 0.3, respectively, based on early experimental studies on GAs [61] and our various experiments. The population size is set to 10, and the threshold value of $|A|$ is set to 100 when evaluating scheduling sequences.

To answer the first three questions proposed in subsection 5.1, the detailed process of implementing BGA, ESS-GA, RUS-GA, and ESS-RUS-GA are provided as follows. BGA, ESS-GA and RUS-GA are illustrated in Algorithms 2, 3 and 4. Additionally, when test data that cover the target path have been generated or the

Algorithm 2 BGA

Require: a program, ***P***, and a target path, p^*.

1: $p_{s_i} \leftarrow 0.5$.
2: Generate an initial population, $pop(1)$.
3: **for** k from 1 to Count, step 1 **do**
4: Calculate the fitness value of each individual $pop(k)$.
5: **if** a test datum that covers p^* is generated **then**
6: Print the test datum.
7: Break.
8: **else**
9: Conduct selection in $pop(k)$.
10: Conduct crossover and mutation with p_{s_i}.
11: **end if**
12: **end for**

Algorithm 3 ESS-GA

Require: a program, ***P***, and a target path, p^*.

1: $A \leftarrow \emptyset$, $p_{s_i} \leftarrow 0.5$.
2: Generate an initial population, $pop(1)$.
3: **for** k from 1 to Count, step 1 **do**
4: Calculate the fitness value of each individual in $pop(k)$.
5: **if** a test datum that covers the target path is generated **then**
6: Print the test datum.
7: break.
8: **else**
9: Conduct selection in $pop(k)$.
10: Put the individuals in $pop(k)$ in archive, A.
11: **if** the threshold value of A is achieved **then**
12: Calculate $F(\{s_i\})$.
13: $p_{s_i} \leftarrow 1 - F(\{s_i\})$.
14: $A \leftarrow \emptyset$.
15: **end if**
16: Conduct crossover and mutation based on the updated p_{s_i}.
17: **end if**
18: **end for**

Algorithm 4 RUS-GA

Require: a program, ***P***, and a target path, p^*.

```
1:  p_{s_i} ← 0.5.
2:  Generate an initial population, pop(1).
3:  for k from 1 to Count, step 1 do
4:      Calculate the fitness value of each individual in pop(k).
5:      if a test datum that covers p* is generated then
6:          Print the test datum.
7:          Break.
8:      else
9:          Conduct selection in pop(k).
10:         Conduct crossover and mutation with the probability of p_{s_i}.
11:         if crossover(mutation) is performed at the scheduling sequence then
12:             Locate the point of crossover(mutation) operations.
13:         end if
14:     end if
15: end for
```

maximal number of generations set in advance has been reached, the algorithm will be ended and the obtained test data will be output.

(1) BGA

The initial population is first generated (Line 2). Based on the similarity between the target path and the one traversed by an individual, the fitness of each individual is calculated (Line 4). The individuals with high fitness are reserved by selection. The location of the crossover and mutation points with an equal probability in the scheduling sequence and the program input is selected, forming two sets of individuals that the crossover and mutation points are located either in the scheduling sequence or in the program input. The single-point crossover and mutation operations are conducted on these two sets to generate the offspring population (Lines 8-10).

(2) ESS-GA

ESS-GA is first employed to evolve the population for a number of generations, until the number of individuals in A has achieved the threshold value (Lines 10). Following that, based on these individuals, each scheduling sequence is evaluated, and the probability of the crossover and mutation points located either in the scheduling sequence or in the program input is calculated. During the evolution, once the number of new individuals in A achieves the threshold value, the

probability will accordingly be updated (Lines 11-15).

(3) RUS-GA

Firstly, the fitness of each individual is calculated and the selection operation is conducted. Secondly, the population is divided into two sets, with the same process as that in BGA and the crossover and mutation operations are conducted on these two sets. For the set of individuals that the crossover and mutation points are located in the scheduling sequence, the points are selected in the encoding corresponding to the WRNGs that affect uncovered nodes (Lines 11-12).

(4) ESS-RUS-GA

When generating test data that cover the target path by using BGA, the process of determining the probability of the crossover and mutation points being located in a region is the same as the approach in ESS-GA. Further, if the crossover and mutation points are located in the scheduling sequence, the process of narrowing down the range of the crossover and mutation points is the same as the strategy in RUS-GA.

5.4. Implementation Details and Indicators

In order to force a program to be under a scheduling sequence, the program is tackled as follows. For WRNGs, MPI_ANY_SOURCE in each receiving statement is replaced with a variable whose value is from the scheduling sequence of a decoded individual. In this way, the source of message received by a statement is specified by an individual during test data generation.

For the path diversity, branch coverage is employed as a criterion. The number of target paths for each program is listed in Table 3. If a path is likely to be infeasible, it is deleted and another one is selected with its feasibility being further judged, until the target number of feasible paths are selected, so that test data that cover these paths cover all its branches. Consequently, these paths have various characteristics in terms of the complexity of a branch condition, the path length, the number of branches, and the difficulty in test data generation, which is beneficial to keeping impartiality in path selection. In addition, this paper emphasizes the problem of test data generation and proposes an enhanced GA based on feedback information during the evolution to tackle this problem. Although branch coverage is accomplished in the experiments, other issues about branch coverage are beyond the scope of this study.

For the proposed method, the overhead of evolutionary test data generation mainly includes calculating the fitness of each individual, evaluating scheduling sequences, and calculating the probability of the crossover and mutation points. In

order to reflect these overheads, we employ the time consumption and the number of generations to evaluate the proposed method and comparative ones. The number of generations means the number that a population has evolved before generating test data that cover a path. Accordingly, the time consumption refers to time spent in evolving the population from the beginning to finding the desired test data. The less the time consumption and the smaller the number of generations of a method are, the higher its efficiency is.

Furthermore, to better explain the experimental results, the reduction rate is utilized to reflect the relative difference between methods and defined as follows.

$$R_g(A,B) = \frac{g_A - g_B}{g_A} \times 100\% \tag{5}$$

$$R_t(A,B) = \frac{t_A - t_B}{t_A} \times 100\% \tag{6}$$

where g and t mean the number of generations and the time consumption, respectively, R_g and R_t are the reduction rate in the number of generations and the time consumption, respectively.

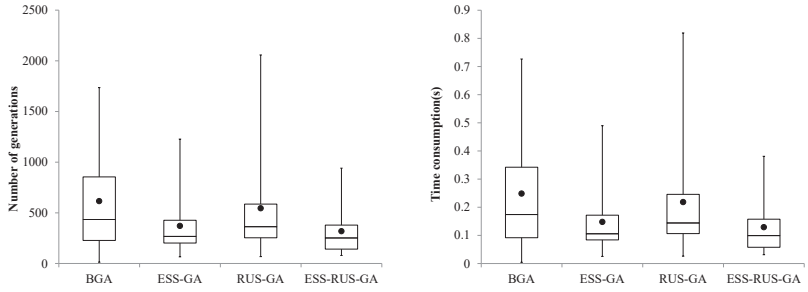
5.5. Experimental Results and Analysis

For each program under test, the maximal number of generations is set to 10000. For each target path, each method is run 20 times independently. For all the target paths of each object, the sum of the number of generations and the sum of the time consumption are calculated against the corresponding run. Figures 3-12 show the number of generations and the time consumption of ten programs under test in the form of box-plot graphs. The values of R_g and R_t of different methods are listed in Tables 4 and 5.

Additionally, the results of the hypothesis testing between each pair of comparative methods are listed in Tables 6, 7 and 8 where'$+$' represents that the proposed method is significantly different from the compared method, and '$=$' means that two methods have no significant difference.

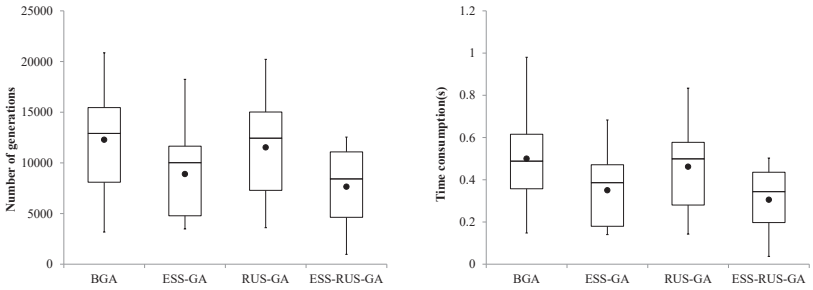
(1) Regarding Q1

ESS-GA consumes a smaller number of generations and less time to generate test data than BGA. *Creator_consumer* has the minimal reduction rate of 5.0% in the number of generations whereas R_g(BGA, ESS-GA) achieves 41.5% for *Matrix*. Meanwhile, *Matrix* also has the maximal R_t(BGA, ESS-GA) of 43.7%, whereas *ASCII* and *Integrate_mw* have the minimal reduction rate of 7.8% in the time consumption. Table 6 reports that there are five programs having the symbol



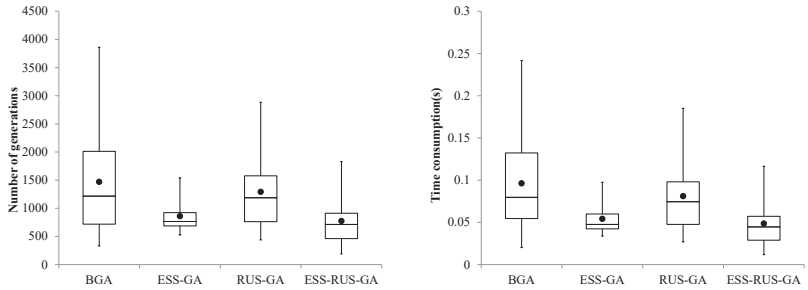
(a) The number of generations (b) The time consumption

Figure 3: Max_triangle



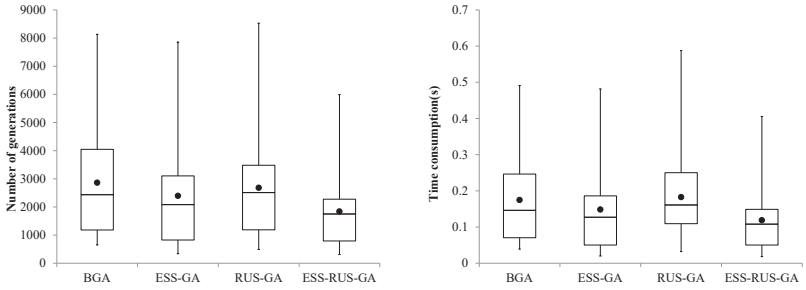
(a) The number of generations (b) The time consumption

Figure 4: Including



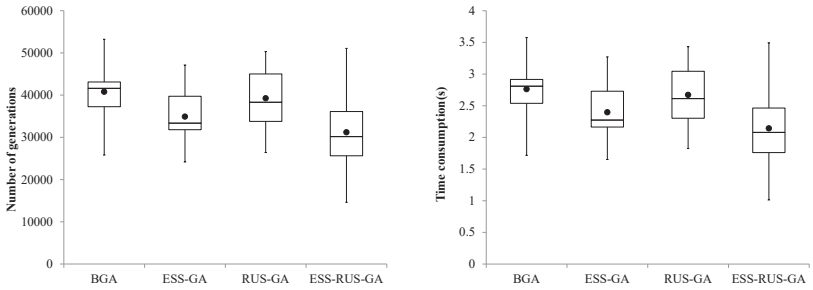
(a) The number of generations (b) The time consumption

Figure 5: Matrix



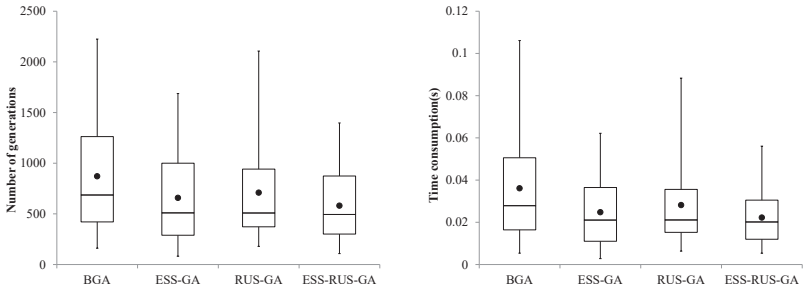
(a) The number of generations (b) The time consumption

Figure 6: Index



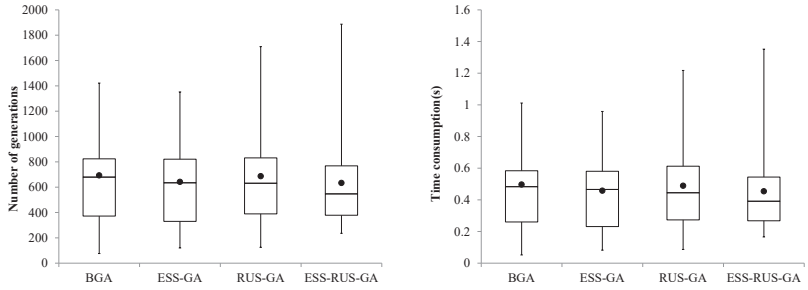
(a) The number of generations (b) The time consumption

Figure 7: Min



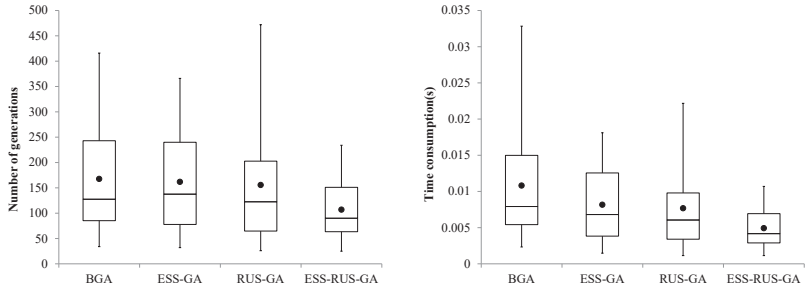
(a) The number of generations (b) The time consumption

Figure 8: Convex_quadrilateral



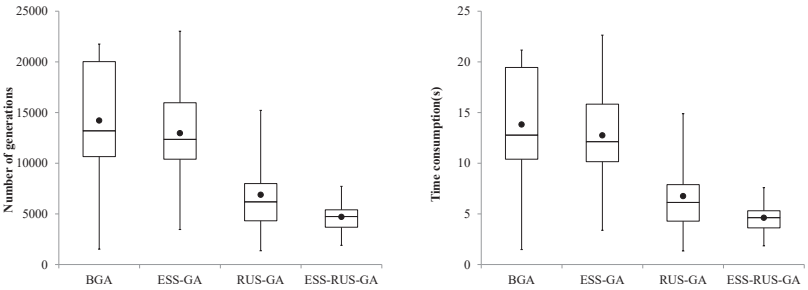
(a) The number of generations (b) The time consumption

Figure 9: ASCII



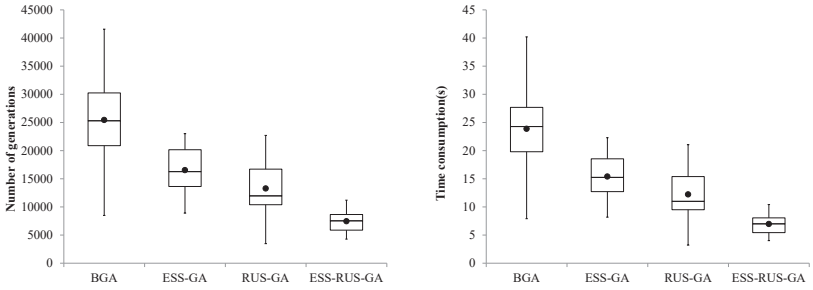
(a) The number of generations (b) The time consumption

Figure 10: Creator_consumer



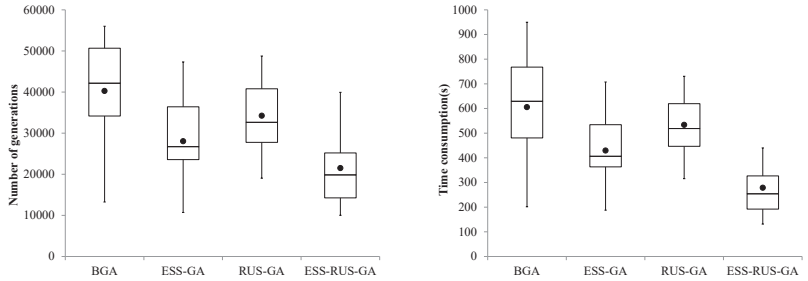
(a) The number of generations (b) The time consumption

Figure 11: Integrate_mw



(a) The number of generations (b) The time consumption

Figure 12: Search_function



(a) The number of generations (b) The time consumption

Figure 13: Kfray

Table 4: R_g of each pair of comparative methods(%).

	(BGA, ESS-GA)	(BGA, RUS-GA)	(BGA, ESS-RUS-GA)	(ESS-GA, ESS-RUS-GA)	(RUS-GA, ESS-RUS-GA)
Max_triangle	39.8	11.5	48.3	14.1	41.5
Including	27.6	6.2	37.8	14.1	33.7
Matrix	41.5	12.1	47.4	10.0	40.1
Index	16.3	6.3	35.8	23.2	31.4
Min	14.4	3.8	23.5	10.6	20.5
Convex_quadrilateral	24.4	18.6	33.3	11.7	18.1
ASCII	7.4	0.9	8.6	1.3	7.7
Creator_consumer	5.0	8.7	12.0	7.3	3.6
Integrate_mw	8.8	51.5	66.9	63.7	31.7
Search_function	35.0	47.8	70.7	54.9	43.8
Kfray	30.4	15.0	46.6	23.3	37.2

Table 5: R_t of each pair of comparative methods(%).

	(BGA, ESS-GA)	(BGA, RUS-GA)	(BGA, ESS-RUS-GA)	(ESS-GA, ESS-RUS-GA)	(RUS-GA, ESS-RUS-GA)
Max_triangle	40.5	12.2	48.1	12.7	40.9
Including	30.0	7.6	39.0	12.8	33.9
Matrix	43.7	15.8	49.5	10.3	40.0
Index	15.0	-4.6	31.9	19.9	34.9
Min	13.2	3.3	22.4	10.6	19.8
Convex_quadrilateral	31.3	22.0	38.4	10.3	21.0
ASCII	7.8	1.5	8.6	0.8	7.2
Creator_consumer	22.8	30.5	31.8	11.6	1.9
Integrate_mw	7.8	51.1	66.6	63.8	31.7
Search˙function	35.5	48.8	70.8	54.8	43.0
Kfray	29.1	11.9	54.0	35.1	47.8

Table 6: The results of the Mann-Whitney U test on the number of generations and the time consumption comparing ESS-GA with BGA.

	Max_triangle	Including	Matrix	Index	Min	Convex _quadrilateral	ASCII	Creator _consumer	Integrate_mw	Search _function	Kfray
Time consumption	=	+	+	=	+	=	=	=	=	+	+
# of generations	=	+	+	=	+	=	=	=	=	+	+

Table 7: The results of the Mann-Whitney U test on the number of generations and the time consumption comparing RUS-GA with BGA.

	Max_triangle	Including	Matrix	Index	Min	Convex _quadrilateral	ASCII	Creator _consumer	Integrate_mw	Search _function	Kfray
Time consumption	=	=	=	=	=	=	=	=	+	+	=
# of generations	=	=	=	=	=	=	=	=	+	+	=

Table 8: The results of the Mann-Whitney U test on the number of generations and the time consumption comparing ESS-RUS-GA with each of BGA, ESS-GA and RUS-GA.

		Max_ triangle	Including	Matrix	Index	Min	Convex_ quadrilateral	ASCII	Creator_ consumer	Integrate _mw	Search_ function	kfray
Time consumption	BGA	=	+	+	=	+	=	=	+	+	+	+
	ESS-GA	=	=	=	=	=	=	=	=	+	+	+
	RUS-GA	=	+	+	+	+	=	=	=	=	+	+
# of generations	BGA	+	+	+	=	+	=	=	=	+	+	+
	ESS-GA	=	=	=	=	=	=	=	=	+	+	+
	RUS-GA	=	+	+	+	+	=	=	=	=	+	+

'+' and six programs having the symbol '=' in the two indicators. Based on the above analyses, ESS-GA is significantly better than BGA for some programs, and has no significant difference with BGA for the others.

(2) Regarding Q2

Compared with BGA, RUS-GA has a smaller number of generations and a less time consumption. Among the ten programs, *Integrate_mw* has the maximal R_g(BGA, RUS-GA) and R_t(BGA, RUS-GA) of 51.5% and 51.1%, respectively. In addition, the reduction rate of -4.6% shows that the average time consumption of BGA is slightly less than that of RUS-GA for *Index*. Table 7 shows that RUS-GA is significantly better than BGA for two programs.

The experimental results show that ESS-GA and RUS-GA can achieve a relatively small reduction of the number of generations and the time consumption, and the improvement in the efficiency of test data generation for path coverage is limited.

(3) Regarding Q3

For all the programs, the average of R_g(ESS-GA, ESS-RUS-GA) and R_t(ESS-GA, ESS-RUS-GA) are 21.3% and 22.1%, respectively. Compared with RUS-GA, ESS-RUS-GA averagely reduces the number of generation and time consumption by 28.1% and 29.3%. For BGA and ESS-RUS-GA, the maximal R_g and R_t are 70.7% and 70.8%, respectively, which corresponds to *Search_function*. With respect to *ASCII*, R_g(BGA, ESS-RUS-GA) and R_t(BGA, ESS-RUS-GA) of 8.6% is minimal. For all the programs, the average of R_g(BGA, ESS-RUS-GA) and R_t(BGA, ESS-RUS-GA) are 39.2% and 42.0%, respectively. Table 8 reports that ESS-RUS-GA is significantly better than BGA for eight programs in terms of the number of generations or the time consumption, and ESS-RUS-GA is statistically superior to ESS-GA and RUS-GA for three and six programs, respectively.

(4) Regarding Q4

Two static methods are implemented based on classical symbolic execution [31] and model checking [62] to generate test data for path coverage of parallel programs.

In order to generate test data based on symbolic execution, one path of each program is first selected. The scheduling sequence is randomly selected for each target path. Whether the path can be covered or not under a scheduling sequence is determined by manual analysis. Then, for scheduling sequences under which the path can be traversed when taking an input, the program input is represented by symbolic values, and the program is symbolically executed to cover the path. Finally, constraints for covering the path are collected and solved to produce test data.

The experimental results of the symbolic execution method are listed in Table 9, which reports that the number of branches in the target paths ranges from 23 to 412, and the number of symbolic values is the same as that of components in the program input domain with the maximum of 125 and the minimum of 2. In addition, the largest number of constraints is 532. Although the method can generate test data, it is inferior to ESS-RUS-GA in terms of the success rate, due to the fact that program execution is closely related to scheduling sequences, and it is difficult for a path to be covered under any scheduling sequence. More specifically, the symbolic execution method can successfully generate test data for all target paths of two programs, whereas ESS-RUS-GA reaches the success rate of 100% for nine programs.

Table 9: The comparison results of ESS-RUS-GA and the symbolic execution method.

Programs under test	# of branches in all target paths	# of symbolic values	# of constraints	Success rate of symbolic execution(%)	Success rate of ESS-RUS-GA(%)
Max_triangle	23	4	31	100	100
Including	35	2	56	56.5	98.6
Matrix	28	16	34	72.3	100
Index	48	15	62	43.6	100
Min	41	125	53	57.1	75.8
Convex_quadrilateral	53	4	64	100	100
ASCII	45	10	71	43.6	100
Creator_consumer	31	2	58	75.5	100
Integrate_mw	102	2	165	46.4	100
Search_function	278	2	356	51.4	100
Kfray	412	20	532	41.6	100

ISP [40], a model checker for MPI programs, takes all scheduling sequences into consideration when detecting deadlock. For each target path, 10000 inputs are randomly sampled, and ISP is executed repeatedly with these inputs to verify whether a path can be covered. It is worth noting that ISP cannot directly produce test data for path coverage. As a result, we actually resort to ISP and random program inputs to traverse target paths. Therefore, essentially, we combine the random method and ISP when covering a path. Correspondingly, the method is denoted as ISP-R. The experimental results of the model checking method are listed in Table 10, where the third and last columns represent the best success rate and time consumption of ESS-RUS-GA in 20 runs. It can be seen that the model checking method has a smaller success rate and larger time consumption than ESS-RUS-GA.

Based on the above experimental results and analysis, it is rational to draw the following conclusions: (1) both ESS-GA and RUS-GA can improve the efficiency in generating test data, and (2) with both methods combined, the enhanced GA

has a significant improvement in efficiency when generating test data. In addition, the comparisons between ESS-RUS-GA and the two static methods support that ESS-RUS-GA is more effective for path coverage of parallel programs.

Table 10: The comparison results of ESS-RUS-GA and the model checking method.

Programs under test	Success rate of ISP-R(%)	Success rate of ESS-RUS-GA(%)	Time consumption of ISP-R(s)	Time consumption of ESS-RUS-GA(s)
Max_triangle	60.0	100	2.81	0.38
Including	71.4	85.7	4.12	0.53
Matrix	100	100	0.79	0.12
Index	100	100	2.06	0.41
Min	33.3	50.0	29.50	3.50
Convex_quadrilateral	100	100	3.43	0.06
ASCII	83.3	100	7.53	1.35
Creator_consumer	100	100	2.33	0.01
Integrate_mw	75.0	100	71.49	7.59
Search_ function	69.2	100	224.19	10.42
Kfray	59.1	100	8175.47	440.24

5.6. Threats to Validity

This section discusses the threats to the validity of experimental results from internal and external factors.

The threats to internal validity depend on parameter settings in methods. GA is intimately related to the crossover and mutation operators, the population size, as well as the maximal number of generations. In order to fairly compare different methods, we set the same parameters values for all the methods. It should be noted that the threshold value of the archive also affects the performance of the proposed method. Therefore, further studies are needed in the future to determine the optimal values of corresponding parameters.

Threats to external validity deal with the limitation in generalizing the experimental results. In order to minimize these threats, we select 11 programs for the experimental study, which have various numbers of processes, communication statements, wildcard receiving statements, WRNGS and lines of code. Furthermore, given the fact that the performance of a test data generation method may be related to target paths, a number of feasible paths are selected as the target ones. Even though they have various characteristics, test data generation methods may still not be fully evaluated by the selected paths. Moreover, the experimental platforms have an influence on the experimental results. To eliminate these threats, we adopt the same hardware and software platforms to implement each method for each program under test. In addition, the evaluation indicators of comparative

methods are another threat to external validity. Therefore, we select the time consumption and the number of generations which are straightforward and intuitive for the performance comparisons. The threat is further tackled by running each method multiple times independently and performing a hypothesis test.

6. Conclusions

For the problem of path coverage of message-passing parallel programs, this paper proposes a feedback-directed genetic algorithm to improve the efficiency of test data generation. The experimental results show that the enhanced GA is very efficient in generating test data for path coverage. More importantly, this paper provides a new perspective to enhance GAs and to address other issues in software testing.

This paper has the following contributions: (1) presenting an approach that uses information provided by a population's evolution to evaluate a scheduling sequence; (2) proposing a strategy that narrows down the search space of the scheduling sequence by the domain knowledge to further improve the GA.

Note that the present study focuses on the problem of one-path coverage for parallel programs, i.e. generating test data to cover only one path in a run. The problem of multi-path coverage can be tackled by running the proposed method multiple times. Alternatively, the problem of multi-path coverage can be formulated as a multi-objective optimization problem. Improved GAs could be developed to solve the problem, so as to generate test data that cover multiple paths in a run, which will be further investigated in our future work. In addition, we intend to explore new methods on path generation and selection.

Acknowledgment

This paper is jointly supported by National Natural Science Foundation of China with grant No. 61773384 and 61503220, and National Basic Research Program of China (973Program) with grant No. 2014CB046306-2, National Key R&D Program of China with grant No. 2018YFB1003802-01.

References

References

[1] N. Fenton, J. Bieman, Software metrics: a rigorous and practical approach, CRC Press, 2014.

[2] S. Scalabrino, G. Grano, D. Di Nucci, M. Guerra, A. De Lucia, H. C. Gall, R. Oliveto, Ocelot: a search-based test-data generation tool for c., in: ASE, 2018, pp. 868–871.

[3] S. R. Souza, M. A. Brito, R. A. Silva, P. S. Souza, E. Zaluska, Research in concurrent software testing: a systematic review, in: Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, ACM, 2011, pp. 1–5.

[4] T. Tian, D. Gong, Evolutionary generation approach of test data for multiple paths coverage of message-passing parallel programs, Chinese Journal of Electronics 23 (2) (2014) 291–296.

[5] Z. Du, S. Li, Y. Chen, P. Liu, Parallel programming technology of high property computing–MPI parallel program design, Beijing: Tsinghua University Press, 2001.

[6] X. Xie, B. Xu, L. Shi, C. Nie, Genetic test case generation for path-oriented testing, Journal of Software 20 (12) (2009) 3117–3136.

[7] M. A. Ahmed, F. Ali, Multiple-path testing for cross site scripting using genetic algorithms, Journal of Systems Architecture 64 (2016) 50–62.

[8] M. Khari, A. Sinha, E. Verdu, R. G. Crespo, Performance analysis of six meta-heuristic algorithms over automated test suite generation for path coverage-based optimization, Soft Computing (2019) 1–18.

[9] X. Yao, D. Gong, G. Zhang, Constrained multi-objective test data generation based on set evolution, IET Software 9 (4) (2015) 103–108.

[10] D. B. Mishra, A. A. Acharya, R. Mishra, Evolutionary algorithms for path coverage test data generation and optimization: A review, Indonesian Journal of Electrical Engineering and Computer Science 15 (1) (2019) 504–510.

[11] S. Xanthakis, C. Ellis, C. Skourlas, A. L. Gal, S. Katsikas, K. Karapoulios, Application of genetic algorithms to software testing, in: Proceedings of the 5th International Conference on Software Engineering and its Applications, 1992, pp. 625–636.

[12] B. Li, Z. Li, Y. Chen, B. Li, Automatic test data generation tool based on genetic simulated annealing algorithm, in: International Conference on Computational Intelligence and Security Workshops, 2008, pp. 183–186.

[13] Y. Jia, W. Chen, J. Zhang, J. Li, Generating software test data by particle swarm optimization, in: Asia-Pacific Conference on Simulated Evolution and Learning, Springer, 2014, pp. 37–47.

[14] X. Yang, Nature-inspired metaheuristic algorithms, Luniver press, 2010.

[15] H. Wang, W. Wang, X. Zhou, H. Sun, J. Zhao, X. Yu, Z. Cui, Firefly algorithm with neighborhood attraction, Information Sciences 382 (2017) 374–387.

[16] H. Wang, W. Wang, L. Cui, H. Sun, J. Zhao, Y. Wang, Y. Xue, A hybrid multi-objective firefly algorithm for big data optimization, Applied Soft Computing 69 (2018) 806–815.

[17] A. Pandey, S. Banerjee, Test suite optimization using firefly and genetic algorithm, International Journal of Software Science and Computational Intelligence (IJSSCI) 11 (1) (2019) 31–46.

[18] M. Mareli, B. Twala, An adaptive cuckoo search algorithm for optimisation, Applied computing and informatics 14 (2) (2018) 107–115.

[19] S. Sheoran, N. Mittal, A. Gelbukh, Artificial bee colony algorithm in data flow testing for optimal test suite generation, International Journal of System Assurance Engineering and Management (2019) 1–10.

[20] Y. A. Alsariera, H. A. S. Ahmed, H. S. Alamri, M. A. Majid, K. Z. Zamli, A bat-inspired testing strategy for generating constraints pairwise test suite, Advanced Science Letters 24 (10) (2018) 7245–7250.

[21] M. W. Przewozniczek, R. Datta, K. Walkowiak, M. M. Komarnicki, Splitting the fitness and penalty factor for temporal diversity increase in practical problem solving, Expert Systems with Applications 145 (2020) 113–126.

[22] A. Chehouri, R. Younes, J. Perron, A. Ilinca, A constraint-handling technique for genetic algorithms using a violation factor, Journal of Computer Science 12 (7) (2016) 350–362.

[23] B. Barney, Message Passing Interface (MPI), `https://computing.llnl.gov/tutorials/mpi/` (2017).

[24] B. Barker, Message passing interface (mpi), in: Workshop: High Performance Computing on Stampede, Vol. 262, 2015.

[25] B. Sun, J. Wang, D. Gong, T. Tian, Scheduling sequence selection for generating test data to cover paths of mpi programs, Information and Software Technology 114 (2019) 190–203.

[26] F. A. Bianchi, A. Margara, M. Pezzè, A survey of recent trends in testing concurrent software systems, IEEE Transactions on Software Engineering 44 (8) (2017) 747–783.

[27] S. M. Melo, J. C. Carver, P. S. Souza, S. R. Souza, Empirical research on concurrent software testing: A systematic mapping study, Information and Software Technology 105 (2019) 226–251.

[28] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, D. Marinov, Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code, in: 2012 34th International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 727–737.

[29] V. Terragni, S. C. Cheung, Coverage-driven test code generation for concurrent classes, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, 2016, pp. 1121–1132.

[30] A. Choudhary, S. Lu, M. Pradel, Efficient detection of thread safety violations via coverage-guided generation of concurrent tests, in: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE, 2017, pp. 266–277.

[31] J. C. King, Symbolic execution and program testing, Communications of the Acm 19 (7) (1976) 385–394.

[32] S. Guo, M. Kusano, C. Wang, Z. Yang, A. Gupta, Assertion guided symbolic execution of multithreaded programs, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 854–865.

[33] S. Guo, M. Kusano, C. Wang, Conc-iSE: Incremental symbolic execution of concurrent software, in: Proceedings of IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 531–542.

[34] X. Zhang, Z. Yang, Q. Zheng, P. Liu, J. Chang, Y. Hao, T. Liu, Automated testing of definition-use data flow for multithreaded programs, in: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2017, pp. 172–183.

[35] D. Khanna, S. Sharma, C. Rodríguez, R. Purandare, Dynamic symbolic verification of mpi programs, in: International Symposium on Formal Methods, Springer, 2018, pp. 466–484.

[36] Y. Lei, R. H. Carver, Reachability testing of concurrent programs, IEEE Transactions on Software Engineering 32 (6) (2006) 382–403.

[37] R. H. Carver, Y. Lei, Distributed reachability testing of concurrent programs, Concurrency and Computation: Practice and Experience 22 (18) (2010) 2445–2466.

[38] X. Qi, J. He, P. Wang, H. Zhou, Variable strength combinatorial testing of concurrent programs, Frontiers of Computer Science 10 (4) (2016) 631–643.

[39] X. Qi, Y. Li, Parallel reachability testing based on hadoop mapreduce, in: International Conference on Software Analysis, Testing, and Evolution, Springer, 2018, pp. 173–184.

[40] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, R. M. Kirby, ISP: a tool for model checking MPI programs, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, 2008, pp. 285–286.

[41] W. Leungwattanakit, C. Artho, H. M, Modular software model checking for distributed systems, IEEE Transactions on Software Engineering 40 (5) (2014) 483–501.

[42] J. S. Vetter, B. R. Supinski, Dynamic software testing of MPI applications with Umpire, in: Proceedings of ACM/IEEE Conference on Supercomputing, 2000, pp. 51–51.

[43] M. Y. Park, S. J. Shim, Y. K. Jun, H. R. Park, MPIRace-Check: detection of message races in MPI programs, in: Proceedings of International Conference on Grid and Pervasive Computing, 2007, pp. 322–333.

[44] B. Krammer, M. M. Resch, Correctness checking of mpi one-sided communication using marmot, in: European Parallel Virtual Machine/Message Passing Interface Users‘ Group Meeting, Springer, 2006, pp. 105–114.

[45] M. S. Ayub, W. U. Rehman, J. H. Siddiqui, Experience report: Verifying MPI Java programs using software model checking, in: 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2017, pp. 294–304.

[46] D. W. Gong, C. Zhang, T. Tian, Z. Li, Reducing scheduling sequences of message-passing parallel programs, Information and Software Technology 80 (2016) 217–230.

[47] K. Sastry, D. E. Goldberg, G. Kendall, Genetic algorithms, in: Search methodologies, Springer, 2014, pp. 93–117.

[48] P. Du, Y. Chu, The improved genetic algorithms apply on parameter estimation of two parameters logistic model on item response theory, Advanced Materials Research 756-759 (2013) 2620–2624.

[49] T. Tian, D. W. Gong, Test data generation for path coverage of message-passing parallel programs based on co-evolutionary genetic algorithms, Automated Software Engineering 23 (3) (2016) 469–500.

[50] L. Cao, W. Zheng, D. Hu, H. Bai, Concurrent program semantic mutation testing based on abstract memory model, in: 2015 IEEE International Conference on Information and Automation, IEEE, 2015, pp. 1200–1205.

[51] A. S. Ghiduk, S. El-Zoghdy, Chomk: Concurrent higher-order mutants killing using genetic algorithm, Arabian Journal for Science and Engineering 43 (12) (2018) 7907–7922.

[52] R. Anbunathan, A. Basu, Combining genetic algorithm and pairwise testing for optimised test generation from UML ADs, IET Software 13 (5) (2019) 423–433.

[53] C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball, Feedback-directed random test generation, in: 29th International Conference on Software Engineering (ICSE'07), IEEE, 2007, pp. 75–84.

[54] C. Pacheco, S. K. Lahiri, T. Ball, Finding errors in. net with feedback-directed random testing, in: Proceedings of the 2008 international symposium on Software testing and analysis, 2008, pp. 87–96.

[55] T. H. Tan, Y. Xue, M. Chen, J. Sun, Y. Liu, J. S. Dong, Optimizing selection of competing features via feedback-directed evolutionary algorithms, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, 2015, pp. 246–256.

[56] X. Dang, X. Yao, D. Gong, T. Tian, Efficiently generating test data to kill stubborn mutants by dynamically reducing the search domain, IEEE Transactions on Reliability 69 (1) (2020) 334–348.

[57] Q. Luo, A. Nair, M. Grechanik, D. Poshyvanyk, Forepost: Finding performance problems automatically with feedback-directed learning software testing, Empirical Software Engineering 22 (1) (2017) 6–56.

[58] S. F. Siegel, T. K. Zirkel, FEVS: A functional equivalence verification suite for high-performance scientific computing, Mathematics in Computer Science 5 (4) (2011) 427–435.

[59] MPI Exercise, `https://computing.llnl.gov/tutorials/mpi/` (2016).

[60] H. Yu, Combining symbolic execution and model checking to verify mpi programs, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, 2018, pp. 527–530.

[61] M. A. Ahmed, I. Hermadi, GA-based multiple paths test data generator, Computers & Operations Research 35 (10) (2008) 3107–3124.

[62] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, R. Majumdar, Generating tests from counterexamples, in: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, 2004, pp. 326–335.