

# Applying GA with Tabu list for Automatically Generating Test Cases Based on Formal Specification

Yuqin Zhou<sup>1</sup>, Taku Sugihara<sup>2</sup>, and Yuji Sato<sup>1,2</sup>(✉)

<sup>1</sup> Graduate School of Computer and Information Sciences,  
Hosei University, Tokyo, Japan

<sup>2</sup> Department of Computer and Information Sciences,  
Hosei University, Tokyo, Japan  
yuji@k.hosei.ac.jp

**Abstract.** How to generate adequate test cases based on a specification to cover all of the paths in its implementation is a challenge in software testing. This paper presents a new approach to selecting test cases for program testing. The essential idea is to generate a set of test cases based on a given operation specification in pre-post notation, and then apply the genetic algorithm to facilitate the generation of more effective test cases in terms of program path coverage. The principle of GA is discussed and an improvement of the GA through integration with Tabu list is presented. An experiment is conducted to study how the improved GA can be applied and to evaluate its effectiveness. The result shows that our proposed method is more effective than conventional methods and can cover all paths based on formal specification.

**Keywords:** Formal specification · Genetic algorithm · Tabu list

## 1 Introduction

Most of testing projects try to adopt automatic test case generation to improve productivity and coverage rate. An important step in automatic testing is to select appropriate test cases. An effective application of formal specification in practice is facilitating test case generation, and the automation of the program testing process [13]. The automatic specification-based testing (ASBT) is a potentially effective technique for software reliability and attractive to the software industry, which can reduce the cost and time and avoid many human errors at the testing process. More and more companies adopt the formal specification to the requirement analysis or system design, this makes the automatic specification-based testing become practical. ASBT has proposed several approaches that let the test cases be generated only based on specifications. The criteria for determining if test cases satisfy the test condition have been put forward, but it does not ensure that every path in the related program can be traversed. Now the problem is how to ensure the paths in the related program can be traversed completely.

To solve the above problem, “Vibration” method has been proposed [12]. It has improved path coverage rate dramatically, however it still can not ensure to cover all paths in the representative program. There is also a large body of research on specification-based testing. Many projects automated test cases generation from specifications, such as Z specification [1, 2], UML statecharts [3, 4], or ADL specifications [5, 6]. The Korat which is one of the novel frameworks for automated testing of Java Programs, is based on Java predicates [7]. Given a predicate and a bound on the size of its inputs, Korat generates all inputs for which the predicate returns true. Marisa A.S’anchez has examined algebraic-specification based testing with particular attention to the approach reported by Gilles Bernot, Marise Claude Gaudel and Bruno Marre [8]. It did not only focus on the generation of test cases from the initial specification but also the additions and revisions during the development. D.Marinov and S. Khurshid presented a framework for automated testing of Java programs called TestEra [9]. TestEra uses the Alloy Analyzer (AA) [10] to automatically generate method inputs and check correctness of outputs, but it requires programmers to learn a specification language much different than Java. Cheon and Leavens have proposed an automatic translation of JML specifications into test oracles for JUnit [11]. But the programmers have to provide sets of possibilities for all method parameters, which add a burden of test cases generation for programmers.

In this paper, Genetic Algorithm (GA) has been used to address the problem by studying how to ensure the paths in the related program can be traversed completely. Roy P. Pargas, Mary J. Harrold, and Robert R. Peck [18] have already applied standard genetic algorithms to generate a test data for a specific path. On the other hand, we propose improved genetic algorithms to generate test cases for all paths in the related program. A large number of test cases are generated by GA, and the appropriate test cases have been chosen after genetic manipulation to cover all paths in the related program. In addition, we improve the algorithm by integrating the Tabu search method.

The remainder of this paper is organized as follows. Section 2 gives a brief introduction to the formal specification and automatic test cases generation. In Sect. 3, we present how to apply GA to solve this issue and improve the GA using Tabu search method. In Sect. 4, we present the result of the experiments. In Sect. 5, we conclude the study and point out some problems in practice we need to solve in the future.

## 2 Formal Specification and Automatic Test Case Generation

Our work proposed in this paper is based on the automatic specification-based testing(ASBT) method by Liu in [13], it is therefore necessary to introduce briefly Liu’s work for readability of this paper. Automatic specification-based testing (ASBT) is a potentially effective technique for software industry, by avoidance of many human errors during a testing process to save development cost and time significantly. In the formal specification, we let  $S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$  represent

the specification of an operation  $S$ , where  $S_{iv}$  is the set of all input variables whose values are not changed by operation  $S$ ,  $S_{ov}$  is the set of all output variables whose values are produced or updated by the operation, and  $S_{pre}$  and  $S_{post}$  are the pre- and post-condition of  $S$ , respectively. Each formal specification  $S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$  can be transformed into an equivalent functional scenario form (FSF):  $(\sim P_{pre} \wedge C_1 \wedge D_1) \vee (\sim P_{pre} \wedge C_2 \wedge D_2) \vee \dots \vee (\sim P_{pre} \wedge C_n \wedge D_n)$ .  $\sim S_{pre} \wedge C_1 \wedge D_1$  is called a functional scenario.

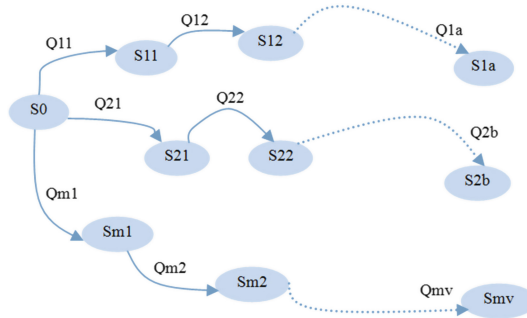
In the functional scenario, the tilde mark  $\sim$  in  $\sim S_{pre}$  represents the initial values of the variables in pre-condition before operation;  $C_i$  is a guard condition which is derived from  $S_{post}$ , which includes only input or initial state variables;  $\sim S_{pre} \wedge C_i$  is test condition. Each test condition can be transformed into an equivalent disjunctive normal form:  $P_1 \vee P_2 \vee \dots \vee P_m$ , each  $P_i$  presents a conjunction of atomic predicates. Each  $P_j$  consists of  $Q_i$ , in the format like this:  $Q_1 \wedge Q_2 \wedge \dots \wedge Q_w$ . To generate a test case based on formal specification is to generate a test case by satisfying the atomic predicates in the test condition. This process can be decomposed into two steps:

**Step1:** Generating a test case  $t$  to satisfy the  $Q_1$

**Step2:** If  $t$  satisfies the all other atomic predicates,  $t$  is a test case; otherwise, go back to *Step1* until finding a test case that satisfies all the atomic predicates or to be stopped by human operation.

The process is illustrated in Fig. 1. Each node denotes a state that satisfies all the predicates expressions along the “path” from the starting state  $S_0$  to itself; and each edge denotes a predication. Therefore, the  $S_{1a}$  represents a state that satisfies all predicates along the “path”, from  $Q_{11}$  to  $Q_{1a}$ . The dotted line in the graph represents omission of many intermediate “state transitions”. All the states  $S_{1a}$ ,  $S_{2b}$ ,  $\dots$ ,  $S_{mv}$  are called accepting states, each of which represents a state that satisfied all the predicates along the path from  $S_0$  to itself. For example,  $S_{1a}$  denotes a state that satisfies the conjunction  $Q_{11} \wedge Q_{12} \wedge \dots \wedge Q_{1a}$  [12, 13].

The essential idea of ASBT is to generate a test set that covers every scenario by satisfying its test condition at least once. Generally, this problem may be handled by generating more test cases, but what test cases should be generated



**Fig. 1.** The graph of the disjunctive normal form of a testing condition

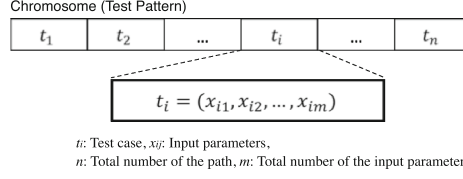
is still a big challenge. Liu has proposed the “Vibration method” which tries to change the “distance” between the variables to find more test cases to cover all corresponding program paths. For example, if there are two expressions E1 and E2, the “distance” ( $|E1 - E2|$ ) between E1 and E2 will be changed as the values of variables in two expressions changed. A test case is created to satisfy the relation when the distance is small; another new test case will be created when the distance is greater. Repeating this process by increasing and decreasing the distance between E1 and E2 until the terminated decision is made. Although this method improved path coverage rate dramatically, it can not ensure that all paths will be covered since the distance is changed randomly at every time. In this paper, we introduce the GA to improve the performance of coverage rate in ASBT. And then we improve the GA using Tabu list, to guarantee a better performance of GA in automatic test cases generation. In next section, we will describe the procedure of how to apply GA to generate test cases based on formal specification [13].

### 3 Applying GA to Automatic Test Case Generation

In this section, we propose applying GA to automatic test case generation based on formal specification. By utilizing GA, the process of choosing test cases will be more effective. Initially, GA generates a population based on the specification, in which each individual is a set of test cases. The individual who has high path coverage rate will be selected to produce next generation. After the GA manipulation, the population in the next generation will be better and have a higher path coverage rate. The evolution will continue until all the paths in the program have been traversed.

#### 3.1 Definition of Chromosome for Automatic Test Case Generation

The first step of GA is to randomly generate population of chromosomes [16]. Each member in the population is called individual. In our work, an individual is a test set consisting of one or more test cases. The test cases are generated based on the formal specification. If the chromosome is defined as a test case, it is difficult to judge whether the chromosome is good enough to be used to generate next generation. Because every chromosome will be in only two states: one is covered a path; the other is not cover a path. If the chromosome is defined a set of test cases, there will be a path coverage rate for every chromosome, and then we can judge which one is appropriate to be selected to generate the next generation. Figure 2 shows an example of a chromosome definition for a set of test cases. Every genetic locus in each chromosome corresponds to a test case, which satisfies the precondition. For example, the input variables are numeric type from 1 to 100, and then genetic locus will be defined as a number from 1 to 100 randomly. The length of the chromosomes (how many test cases in a chromosome) equals to the number of the paths. Thus, in the Fig. 2,  $t_i$  represents a test case, and  $n$  denotes the number of the paths. By generating a large scale of chromosomes in a population, we can generate a lot of test cases satisfied the precondition.



**Fig. 2.** Chromosome definition for test cases generation

Figure 3 shows an example of the initial population when we don't have any previous test patterns or knowledge to generate test patterns from corresponding paths map. In this example, input parameters are generated randomly in a feasible region. Figure 4 shows an example of the initial population when we have some previous test patterns or knowledge to generate test patterns from corresponding paths map. In this example, we use these test patterns as a part of chromosomes and generate other input parameters randomly.

### 3.2 Evaluation Function

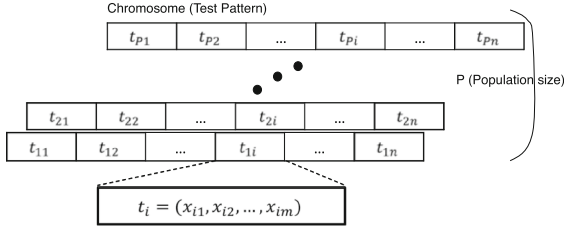
In a genetic operation, the evaluation function is a fitness function which is devised for each problem to be solved. Given a particular chromosome, the fitness function returns a single numerical "fitness", or "figure of merit", which is supposed to be proportional to the "utility" or "ability" of the individual which that chromosome represents. In this case, the fitness function is defined as the path coverage rate, when the chromosome covered more paths, its evaluation value will be higher. In this study, we assume that the number of all paths in the related program is known and define the fitness function in Eq. (1) below, which is normalized to values between 0 to 1.

$$f = \frac{k}{n} \quad (1)$$

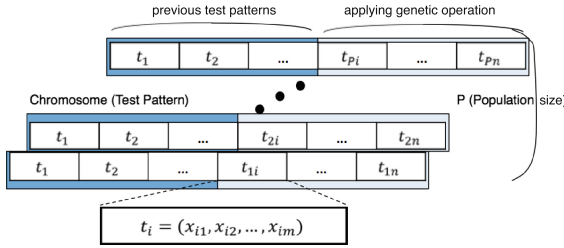
In Eq. (1),  $f$  is the evaluation function,  $k$  is the number of the paths that chromosome has covered;  $n$  is the number of all paths in the corresponding programs. The individual who achieves an evaluation value  $f=1$  means that the individual has covered all paths in the corresponding programs, and it is the optimal solution to the problems.

### 3.3 Genetic Manipulation

**Selection.** Selection is the stage in which individual generates for next generations from a population based on fitness values. There are many kinds of selection methods in GA, and we apply tournament selection [16] in this study. Tournament selection has a simple rule and can guarantee that the better one will be chosen and the worse one will be eliminated. Tournament selection involves randomly picking several individuals from the population and staging a tournament to determine which one is finally selected. It generates a random value between



**Fig. 3.** An example of initial populations when we don't have any previous test patterns

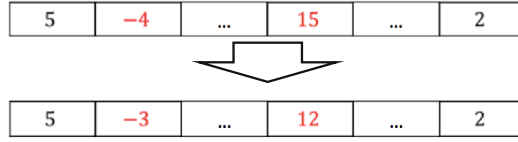


**Fig. 4.** An example of initial populations when we have some previous test patterns

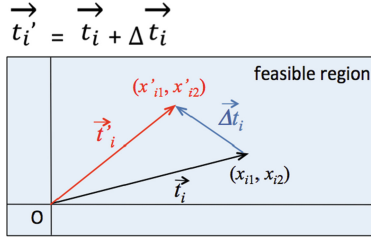
zero and one and comparing it to a pre-determined selection probability. If the random value is less than or equal to the selection probability, the fitter candidates is selected; otherwise, it select other candidates again. The probability parameter provides a convenient mechanism for adjusting the selection pressure. The tournament size is 3, selecting the highest fitness one to be parent for next generation by comparing the three individuals. The individuals selected by tournament selection as parents to do crossover.

**Crossover.** Crossover is a genetic operator used to vary the programming of chromosomes from one generation to the next. In this paper we choose two-point crossover [16] since it can keep the stability of the individuals' value of fitness when the average fitness in the population is high enough. There is a variable  $t$ , which is generated randomly. When it is bigger than the crossover rate, the crossover will take two individuals, and cuts their chromosome strings at some randomly chosen position, to produce two "head" segments, and two "tail" segments. The two segments between the "head" and "tail" are then swapped over to produce two new full length chromosomes. The two offsprings each inherit some genes from each parent.

**Mutation.** Mutation is a genetic operator used to maintain genetic diversity. When a number of the input parameter is 1, we can apply a simple mutation. An example of a simple mutation is shown in Fig. 5. Mutation is applied to each child individually after selection and crossover. It randomly alters each gene with a small probability. If the variable  $t$  which is generated randomly is bigger than



**Fig. 5.** An example of a simple mutation



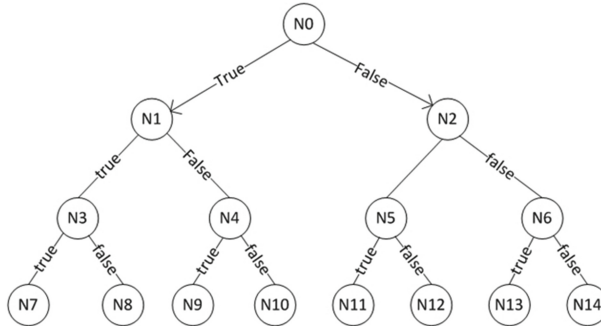
**Fig. 6.** An example of a different vector mutation for the two input parameters

mutation rate, the mutation manipulation will be executed. Mutation provides a small amount of random search, and helps ensure that no point in the search space has a zero probability of being examined. When a test case is consisted of multiple input parameters, a mutation is defined as a different vector mutation. Figure 6 shows an example for the two input parameters.

Repeat the above steps until the individuals whose fitness is 1 have been found or the termination condition has been met [14–16].

### 3.4 Proposal of GA with Tabu List for Covering Paths

We design test cases generation method based on the standard GA to find all paths in the corresponding program. Figure 6 shows an example of the paths in a program, in the form of binary tree. Gene has been designed as a test case,



**Fig. 7.** A path map of a test program

**Record**

$N_0-N_1-N_3-N_7$	$N_0-N_1-N_3-N_8$	$N_0-N_1-N_4-N_9$	$N_0-N_1-N_4-N_{10}$	$N_0-N_2-N_5-N_{11}$	$N_0-N_2-N_5-N_{12}$	$N_0-N_2-N_6-N_{13}$	$N_0-N_2-N_6-N_{14}$
-------------------	-------------------	-------------------	----------------------	----------------------	----------------------	----------------------	----------------------

**Fig. 8.** An array records paths which have been found like a Tabu list

and the number of genes contained in each individual equals to the number of the paths. There are 8 paths in Fig. 7, each node is a predicate extracted from the formal specification. Every individual generated 8 test cases once a time, but the test cases did not cover all paths necessarily. After selection and crossover operation, individuals will cover more and more paths; therefore the population will be better and better. This process will be terminated until the fitness = 1 has been found. For example, there is an array to record paths which has been covered. If one of the test cases covered path  $N_0-N_1-N_3-N_7$ , we add this path to the array, as shown in Fig. 8. The GA will not be terminated until all paths have been recorded. In other words, the process will be stopped before the best individual has been found, which saves the time and makes the algorithm more effective.

## 4 Experiments

### 4.1 Experiment Method

Judging triangle's type is often introduced in the research about software testing as a typical example to demonstrate the idea of the testing. Firstly, we create the formal specification of judging triangle's type including precondition and post-condition, as shown in Fig. 9. There are three inputs and five paths in the program that implements the specification, as shown in Fig. 10. If the three inputs can not form a triangle, it is a test case that covers the path1. If the three inputs can form a triangle, they will cover one of the other four paths. All inputs are real number type. Due to the complexity and clarity of the logical in this program, even there are plenty of combinations of inputs, only a small amount of combinations can cover certain paths in the program. Therefore, blind search became costly.

The experiment environment parameters are shown in Table 1. Then we change the parameters in GA to determinate what the parameters set for. Figure 11 shows

**Table 1.** Experiment environment

OS	Windows 7
Processor	Intel CoreI i7 CPU 2.80 GHz
Memory(RAM)	8 GB
System type	64-bit operation system
Tool	Eclipse

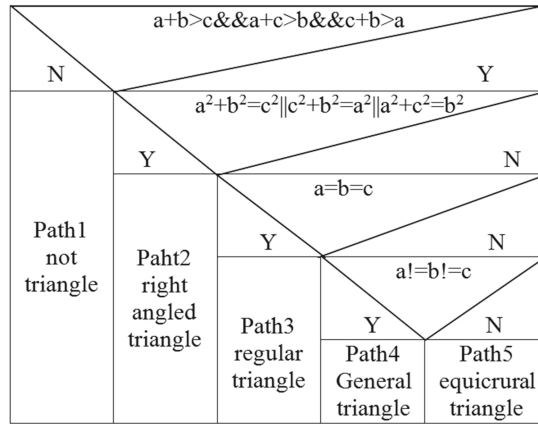


```

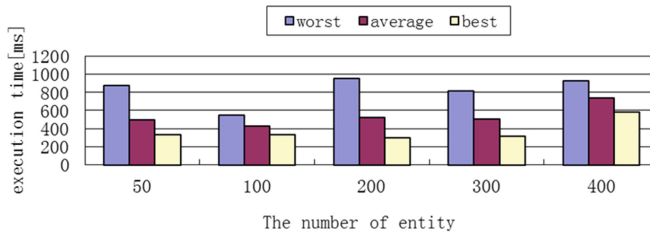
process triangle(x:nat,y:nat,z:nat)T:string
pre    x<=100&&y<=100&&z<=100
post  if x+y>c&&x+z>y&&y+z>x
      then if  $x^2 + y^2 == z^2 \parallel x^2 + z^2 == y^2 \parallel y^2 + z^2 == x^2$ 
        then T="right-angled triangle"
      else if x==y==z
        then T="regular triangle"
      else if x!=y&&y!=z&&x!=z
        then T="general triangle"
      else T="isocetes triangle"
    else T="not a triangle"
end_process

```

**Fig. 9.** An example of a formal specification to judge triangle's type

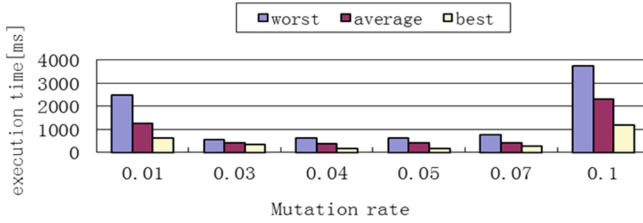


**Fig. 10.** An example of a paths map to judge triangle's type

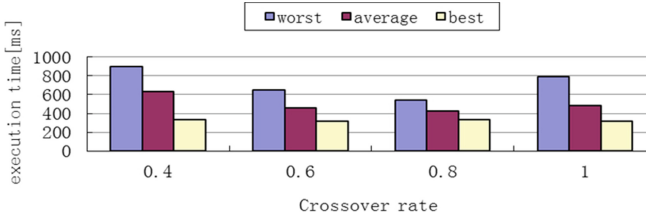


**Fig. 11.** Relationship between the execution time and the number of entity.

that if the number of the population is changed to 50, 100, 200 and 300, respectively, the execution time will be different. As the Fig. 11 shows that while the population size in 100 the experiment results are the stablest. From Fig. 12, we



**Fig. 12.** Relationship between the execution time and a mutation rate.



**Fig. 13.** Relationship between the execution time and a crossover rate.

can find that while the mutation rate is from 0.03 to 0.05 the cost of time is less. And the Fig. 13 shows cost of time is not very sensitive to crossover rate, in contrast, the mutation rate influenced the results more than other parameters.

Table 2 shows the execution parameters. The GA execution parameters are determined by preliminary tests and the choice of selection and crossover method. Based on the standard GA, we don not need to apply complicated selection and crossover method. In this phase, we prove that applying GA to automatically test cases generation is feasible and effective.

The design of the experiments is limited by the type of inputs. When the types of the input are naturel number, real number, or char, the design will be relatively easy. In this experiment, only inputs of naturel number type will be considered.

**Table 2.** GA execution parameters

No. of entities	100
Crossover rate	0.8
Mutation rate	0.03
Selection method	Tournament method
Crossover method	Two-point crossover
Mutation method	Simple mutation
Length of gene	35
Repeat time	50

**Table 3.** Execution time and generation

Execution Time			Generation			Path coverage rate
best	average	worst	best	average	worst	
121 ms	427 ms	940 ms	30	120	268	100 %

## 4.2 Experimental Results

Table 3 shows the results of the execution time and generations until a solution is reached by GA which repeats 50 times and all paths are covered. The results showed in Table 3 that the average generation is around 100, and the average time is less than 0.5 s.

## 4.3 Experiments About Comparison of GA and GA with Tabu List

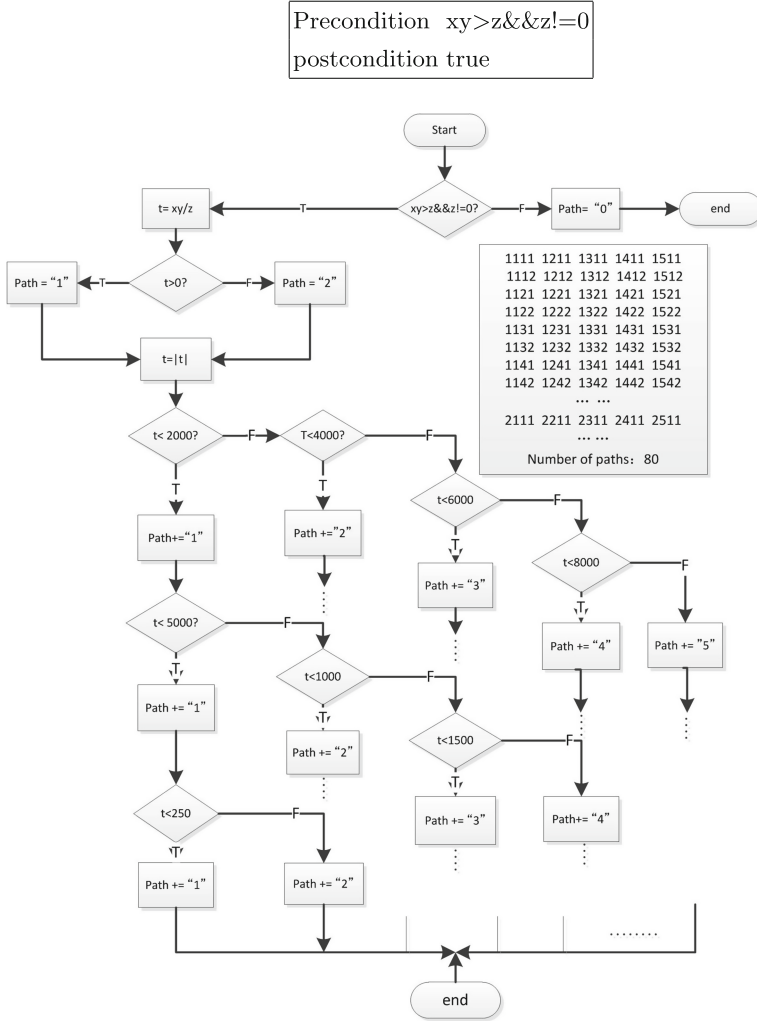
The example containing 80 paths has been introduced here to show the difference between the standard GA and the GA with Tabu list. Figure 14 shows the formal specification and the paths we designed for the experiment.

Based on the formal specification, there are three inputs including variables  $x$ ,  $y$  and  $z$ . However, it is not clear about how many paths in the relative program since the postcondition is true. We design a map of test paths based on the formal specification. In this map, there are 80 paths from start to end. As we marked the node in the map, every path has a sequence number when it passes through the nodes, which is showed at right Fig. 14. We generate test cases to cover the 80 paths in order to cover all paths in the relative program as far as possible. With the standard GA, we need to find the best individual who has covered all paths in the map; by using GA with Tabu list, the process will end when all paths have been traversed.

Table 4 shows the execution parameters in this experiment. The standard GA and GA with Tabu list have different parameters according to their different designs. As a result, when the number of entities is 750, we achieved a best result

**Table 4.** Experiment parameters

	GA	GA with Tabu list
2 No. of entities	750	50
Crossover rate	0.9	0.9
Mutation rate	0.03	0.03
Selection method	Tournament method	Tournament method
Crossover method	Two-point crossover	Two-point crossover
Mutation method	Simple mutation	Simple mutation
Length of gene	80	80
Repeat time	50	50



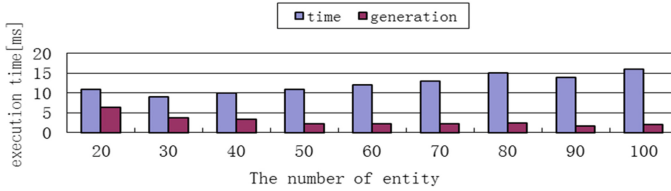
**Fig. 14.** An example of formal specification and a map of test paths

using standard GA; on the other hand, it costs too much time when the number of entities is large in the experiment using GA with Tabu list.

Table 5 shows the results of the two methods. When using standard GA, the execution will be terminated when the individual whose value of fitness is 1 has been found. It takes too much execution time to find the best individual who has covered all paths in the program. In practice, we merely need to cover all paths in the related program. The experiment of GA with Tabu list shows a much better result comparing with the standard GA. The GA with Tabu list has been terminated when all paths have been covered before finding the best one who has a best path coverage rate. From the Table 5, the GA with Tabu list has

**Table 5.** Comparison of GA and GA with Tabu list

	Average time	Average generation
GA	15033 ms	273.36
GA with Tabu list	11 ms	2.78

**Fig. 15.** Relationship between number of entities and execution time in the method using GA with Tabu list

saved much time and generation; that is because it did not cost too much time to find the best individual, and just several generations need to be produced to cover all paths in the related program.

Since the size of the population has influenced the results of the experiment using GA with Tabu list, we have changed the size of the population to do another experiment to see the relationship between the size of the population and the execution time of CPU. As is shown in Fig. 15, it got a best result around the population size of 30.

#### 4.4 Discussion

Applying GA to automatic test cases generation can cover all paths in the test program effectively, as is shown in Table 6. The Vibration method which has been proposed covering all paths in the related program, but the results showed that the coverage rate is 92 % [12]. From Table 6, the V-method(Vibration method) has a much better result than Pairwise testing, however the coverage rate is still not reach the 100 %. The GA can cover all paths based on formal specification which is more effective than V-method and Pairwise testing method. On the other hand, this method need high cost for fitness evaluations of individuals because of the necessity for the running the program many times. Therefore, we need to speed up the proposed method using parallel processing on many-core architectures.

In order to improve the GA, we combined GA with Tabu search, and execution time has been reduced obviously, which is showed in Table 5. As shown in Table 5, the test case generation method using GA with Tabu list takes much less time than the standard GA. In standard GA, we attempt to find the best individual who covered all paths, on the contrary, the GA with Tabu list found all paths in the program before finding the best one. In the beginning, the two

**Table 6.** Comparison of path coverage rate

Testing methods	Number of test cases	Path coverage rate
Pairwise testing	58	53 %
V-Method	56	92 %
Standard GA	80	100 %
GA with Tabu list	80	100 %

methods are introduced to find the test cases which satisfy the test condition, afterwards, the standard GA focus on find the best one who covered all paths, while the GA with Tabu list does not care about the fitness function in the later time.

In this paper, we assume that the number of all paths in the related program is known. In the other hand, in practice, we do not know how many paths in the related program. Fortunately, there is an open source software [17] on the internet which can check out the number of paths in the program. It makes that generating the test cases to cover all paths in the test program with our method become practicable. We also need evaluations using a more complicated and practical program as our future work.

## 5 Conclusion

In this paper, we applied GA to generate test cases based on a formal specification. We demonstrated the procedure of the design about how to generate test cases using GA and our proposed method is more effective than conventional methods and can cover all paths. We also conducted two experiments to test our methods; one is to apply the standard GA to generate test cases of judging a triangle; the other containing 80 paths to compare the performance of GA and GA with Tabu list. The results show that the path coverage rate can reach 100 %, and the GA with Tabu list has a better performance than standard GA. How to locate the bugs in the program is still a problem unaddressed in this paper. Our future work will concentrate on detecting the bugs in the program using a more practical problem.

**Acknowledgements.** First and foremost, we would like to show my deepest gratitude to Prof. Liu, a respectable and responsible professor of the Department of Computer and Information Sciences in Hosei University, for their valuable comments and discussions on experimental results. This research is partly supported by the Hitachi Management Partner Corporation.

## References

1. Horcher, H.-M.: Improving software tests using Z specifications. In: Proceeding 9th International Conference of Z Users, The Z Formal Specification Notation (1995)

2. Spivey, J.M.: *The Z Notation: A Reference Manual*, 2nd edn. Prentice Hall, Englewood Cliffs (1992)
3. Offutt, J., Abdurazik, A.: Generating tests from uml specifications. In: France, R.B. (ed.) *UML 1999*. LNCS, vol. 1723, pp. 416–429. Springer, Heidelberg (1999)
4. Rumbaugh, J., Jacobson, I., Booch, G.: *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series, Boston (1998)
5. Chang, J. Richardson, D.J.: Structural specification-based testing: automated support and experimental evaluation. In: *Proceeding 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 285–302, Sept 1999
6. Sanker, S. Hayes, R.: Specifying and testing software components using ADL. Technical report SMLI TR-94-23, Cun Microsystems Laboratories, Inc., Mountain View, CA, April 1994
7. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automatically testing based on java predicates. In: *ISSTA 2002 Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 123–133 (2002)
8. Marisa A. S'anchez: *Specification-based Testing*. pp. 12–47 1997
9. Marinov, D., Khurshid, S.: TestEra: a novel framework for automated testing of Java programs. In: *Proceeding 16th IEEE International Conference on Automated Software Engineering*, San Diego, CA, Nov. 2001
10. Jackson, D., Schechter, I., Shlyakhter, I.: ALCOA: The alloy constraint analyzer. In: *Proceeding 22nd International Conference on Software Engineering(ICSE)*, Limerick, Ireland, June 2000
11. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. Technical report 01–12, Department of Computer Science, Iowa State University, Nov 2001
12. Liu, S., Nakajima, S.: A vibration method for automatically generating test cases based on formal specifications. In: *Software Engineering Conference, 2011 18th Asia Pacific*, pp. 73–80, 5–8 Dec. 2011
13. Liu, S., Nakajima, S.: A decomposition approach to automatic test cases generation based on formal specification. In: *4th IEEE International Conference on Secure Software Integration and Reliability Improvement*, Singapore, pp. 147–155, IEEE CS Press, 9–11 June 2010
14. Reeves, C.R.: *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell Scientific, Oxford (1993)
15. Beasley, J.E.: A genetic algorithm for the set covering problem. *Eur. J. Oper. Res.* **94**, 392–404 (1996)
16. Beasley, D., Bull, D.R., Martin, R.R.: An overview of genetic algorithms: Part I, fundamentals. *Univ. Comput.* **15**, 58–59 (1993)
17. <http://www.kailesoft.cn> (cited 7.7.2014)
18. Pargas, R.P., Harrold, M.J., Peck, R.R.: Test-Data generation using genetic algorithms. *J. Softw.Test. Verif. Reliab.* **9**, 263–282 (1999)

Structured Object-Oriented Formal Language and  
Method

4th International Workshop, SOFL+MSVL 2014,  
Luxembourg, Luxembourg, November 6, 2014, Revised  
Selected Papers

Liu, S.; Duan, Z. (Eds.)

2015, VIII, 189 p. 74 illus., Softcover

ISBN: 978-3-319-17403-7