An Empirical Evaluation of Test Data Generation Techniques

Seung-Hee Han
Network Technology Laboratory, KT, Korea
shehan@kt.com

Yong-Rae Kwon
Division of Computer Science, Dept. EECS, KAIST, KOREA
kwon@cs.kaist.ac.kr

Received 21 January 2008; Accepted 11 July 2008

Software testing cost can be reduced if the process of testing is automated. However, the test data generation task is still performed mostly by hand although numerous theoretical works have been proposed to automate the process of generating test data and even commercial test data generators appeared on the market. Despite prolific research reports, few attempts have been made to evaluate and characterize those techniques. Therefore, a lot of works have been proposed to automate the process of generating test data. However, there is no overall evaluation and comparison of these techniques. Evaluation and comparison of existing techniques are useful for choosing appropriate approaches for particular applications, and also provide insights into the strengths and weaknesses of current methods. This paper conducts experiments on four representative test data generation techniques and discusses the experimental results. The results of the experiments show that the genetic algorithm (GA)-based test data generation performs the best. However, there are still some weaknesses in the GA-based method. Therefore, we modify the standard GA-based method to cope with these weaknesses. The experiments are carried out to compare the standard GA-based method and two modified versions of the GA-based method.

Categories and Subject Descriptors: Automatic Test Data Generation [**Software Engineering**]

General Terms: Software Testing

Additional Key Words and Phrases: Testing, Test Data, Evaluation

1. INTRODUCTION

Since software testing for assuring software quality is known to account for approximately 50 percent of development cost, how to reduce its cost has always

been a challenge to software engineers [Beizer 1990]. This cost can be reduced if the process of testing is automated. While some parts of testing process are being well served by commercial tools, most works involved in actual generation of test inputs are still performed manually [Meudec 2001].

During the last two decades, various test data generation techniques have been proposed. These can be classified into functional testing and structural testing. Our work focuses on structural testing. We can divide the test data generation approaches based on structural testing into two classes: static approach and dynamic approach. Symbolic execution is a typical example of the static approach. By executing a program symbolically, we can obtain an algebraic expression including symbolic input variables for a given path. In stead of using variable substitution, the dynamic approach executes the program with randomly selected input. Consequently, the values of variables at any time of the execution are used to find adequate test data. The dynamic approach regards the test data generation as a function minimization problem and search for an input which minimizes the specified objective function. A hybrid of the above two approaches has been presented in [Gupta et al. 1998] to combine the advantages of both approaches [Edvardsson 1999].

Although many techniques for automatic test data generation have been developed [Coward 1988; Gallagher and Narasimhan 1997; Gupta et al. 1998; Sthamer 1995; Korel 1990b; Lapierre et al. 1999a; Meudec 2001; Michael et al. 2001], there is no overall evaluation and comparison of these techniques. Evaluation and comparison of existing techniques are useful for choosing appropriate approaches for particular applications. For example, if the high quality of software is essential, a technique with a highest test coverage may be required regardless of cost. On the other hand, if execution time for the test data generation is the most important factor, a fast technique may be desirable, even though some tests that can expose faults may be omitted from the test suite. Evaluation and comparison of existing approaches also provide insights into the strengths and weaknesses of current methods.

This paper conducts experiments on four representative test data generation techniques and discusses on the results according to the following evaluation criteria: **performance**, **quality of test data** and **generality**. In our study we take the random test data generation as the baseline for comparison, and select the iterative relaxation method to represent the hybrid approach [Gupta et al. 1998], and the Korel's method [Korel 1990b; 1990a] and the genetic algorithm (GA)-based technique [Michael et al. 2001; Sthamer 1995] to represent the dynamic approach. There are a lot of technical difficulties associated with symbolic execution in handling input variable-dependent loop conditions, input variable-dependent array references, and module calls [Coward 1988]. However, no practical solutions for these problems are yet available, which prevent the successful application on programs of a general form [Gallagher and Narasimhan 1997]. Therefore, symbolic execution-based technique is excluded from our experiments.

In the performance criterion, the GA-based method had the highest test coverage rate while IRM-based method was a time-efficient method. The Korel's method gave the best quality of test data. Lastly, our experiments evaluated the GA-based method is the most general applicable even if a program to be tested is complex. The overall

result of the experiments have shown that the GA-based technique performs the best. However, there are still some weaknesses in the GA-based method. Although the GA-based method achieved the highest coverage, it required a lot of execution time. Therefore, we introduce the static analysis information into the GA-based method to cope with these weaknesses. To improve the speed of the GA-based method, we can also combine the local optimization algorithm and the GA. We empirically compare the standard GA-based method and two modified versions of the GA-based method.

The main benefit of our study is that it provides a way to evaluate and compare existing test data generation techniques systematically. Based on our experimental results, an improved version of the test data generation algorithms may be derived. The second contribution of our study is that it provides a way to speed up the GA-based method.

The paper begins with introduction of the various test data generation techniques in Section 2. Our empirical study of test data generation techniques is discussed in Section 3. Explanation and experimental results of two modified versions of the GA-based technique are described in Section 4. Finally, in Section 5 conclusions are made and future work is suggested.

2. AUTOMATIC TEST DATA GENERATION

There are two fundamental approaches - functional testing and structural testing- to identifying test cases. In structural testing, the implementation is known and used to identify test cases. Structural testing uses test coverage criteria. Test coverage criteria provide a way to explicitly state the extent to which a *test requirement* (i.e., statements, branches, or conditions) has been tested. We call the program to be tested *the program under test*.

Several techniques to automate the generation of test data can be classified into static approach, the dynamic approach and hybrid approach. In this section, each approach is to be explained and one or two representative test data generation methods of each approach is to be described.

2.1 Random Test Data Generation

In random test data generation, inputs are produced at random until a useful input is found. It is the simplest method of test data generation techniques and is applicable to any type of the programs. However, random test data generation mostly does not perform well in terms of coverage criteria. Since the probability of selecting an input that discovers the semantically small faults can be low. A semantically small fault is a fault that is revealed only by a small percentage of the program input. However, random test data generation is used frequently as a benchmark because it is easy to implement and commonly reported in the literature.

2.2 Static Approach

Static approach does not execute the program under test but generates test data using static information of the program. Symbolic execution is a representative

```
S1: x = y*2;
P1: if (x >= 10) {
S2:     x = x+10;
S3:     y = y+1;
    }
P2: if (x+y < 110)
S4:     write("OK");
S5: ...
```

Figure 1. An example program.

example of static approach. Symbolic execution [Clarke 1976] is executing a program symbolically, which means that variable substitution is used instead of using actual values, providing an algebraic expression of symbolic input variables for a given path. Thus, ideally, test data generation can be reduced to a problem of solving an algebraic expression, which is a path constraint generated by symbolic execution. Many test data generation methods use symbolic execution to find inputs that satisfy a test requirement (e.g., [Lapierre et al. 1999a; Meudec 2001]). As an example, consider the program given in Figure 1. On the path $P <S1, P1, S2, S3, P2, S5>$, if the input variable y is initially assigned the symbol Y, then after execution of the statement $S1$ and the predicate $P1$, the path constraint is "$Y * 2 >= 10$". Similarly, symbolic execution of the assignment statements $S3$ and $S4$ results in "$x = Y * 2 + 10$" and "$y = Y + 1$". The predicate $P2$ must be false not to exercise the statement $S4$. Therefore, the path constraint of the predicate $P2$ becomes "$(Y * 2 + 10) + (Y + 1) >= 110$", which can be simplified to "$Y >= 33$". For programs having many conditionals and loops, the path predicate becomes very long and, therefore, the path constraint is usually rear ranged and simplified to remove redundant conditions. The set of test data which will cause the test path to be executed is found by solving this set of inequalities, and if a solution can be found (i.e., the set is consistent) the test path is feasible and the solution corresponds to the set of input data which will execute the test path. On the other hand, if the system is inconsistent, then the test path is infeasible.

However, a number of problems may be encountered when symbolic execution technique is applied to automatic generation of test data [Meudec 2001]. Some problems are primarily due to features of programming languages. One such problem arises in indefinite loops. Loops which are input variable dependent can be executed any number of times. Array references can be problematic where the index is not a constant but a variable. Other characteristics of structured programming languages, which are difficult to deal with using symbolic execution, are dynamic memory allocation and pointers. Function calls to modules where there is no access to source code can not be handled. Furthermore, symbolic execution also implies that a symbolic evaluator for the particular language must be built, which indeed requires a great amount of work. As another problem, most symbolic executors simply generate all the syntactic paths in a program (with special considerations for loops). Therefore, it is necessary to be able to check the feasibility of the partial path traversal conditions generated to be able to generate actual test data. Unfortunately, the complexity of the path traversal conditions have, up to date, proved too high to be tackled efficiently and automatically.

For the general usage of test data generation techniques, the base method like symbolic execution should be able to handle all language structures. However, those inherent problems prohibit symbolic execution from being applied to the general applications practically. Thereby, we exclude the techniques using symbolic execution from our experiments.

2.3 Dynamic Approach

Instead of using variable substitution, actual execution runs the program with a randomly selected input in the dynamic approach. This paradigm is based on the idea that if some desired test requirement is not satisfied, data collected during execution can be still used to determine which test inputs come closest to satisfying the requirement. With the help of this feedback, inputs are incrementally modied until one of them satisfies the test requirement. For example, suppose that a hypothetical program contains the condition

if ($pos >= 21$) ...

and that the goal is to ensure that the **TRUE** branch of this condition is taken. We must find an input that will cause the variable pos to have a value greater than or equal to 21 when this condition is reached. A simple way to determine the value of pos is to execute the program up to this condition and then record the value of pos.

Let $pos(x)$ denote the value of pos recorded when the program is executed in the input x. Then, the function

$$\Im(x) = \begin{cases} 21 - \text{pos(x)} & \text{if } pos(x) < 21 \\ 0 & \text{otherwise} \end{cases}$$

is minimal when the **TRUE** branch is taken. Thus, the problem of test data generation is reduced to one of *function minimizatio*n: To find the desired input, we must find a value of x that minimizes $\Im(x)$. In a sense, the function $\Im$(which we will also call an objective function) tells the test generator how close it is to reaching its goal. If x is a program input, then the test data generator can evaluate $\Im(x)$ to determine how close x is to satisfying the test requirement currently being targeted. The idea is that the test data generator can now modify x and evaluate the objective function again in order to determine what modifications bring the input closer to meeting the test requirement. The test data generator makes a series of successive modifications to the input value using the objective function for guidance and, hopefully, this leads to an input that satisfies the test requirement.

While the advantages are that we can handle array references, function calls, and dynamic data structure such as pointers, the dynamic approach is quite expensive because it requires many iterations before a suitable input is found. Moreover, the dynamic approach requires additional execution time because of execution of the program under test [Edvardsson 1999].

2.3.1 *Korel's method*

This dynamic test data generation paradigm is exemplied by the TESTGEN

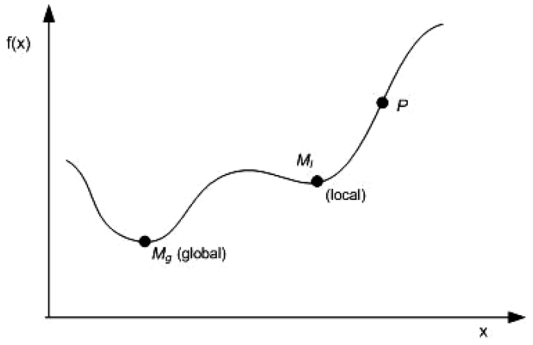

Figure 2. Suppose we start at point P. If we perform gradient descent, the minimum we encounter is the one at M_l, not that at M_g. M_l is called a *local minimum* and corresponds to a partial solution. M_g is the *global minimum* we seek and, unless measures are taken to escape from M_l, M_g will never be reached.

system of [Korel 1990a], as well as [Gallagher and Narasimhan 1997; Michael et al. 2001]. In the Korel's method (TESTGEN system), function minimization method is based on the direct search method for local optimization. For example, suppose we want to take the **TRUE** branch of $P1$ in Figure 1. The input variable y has a initial value of 3. One execution gives x a value of 6, and an objective function $\Im(y)$,

$$\Im(y) = \begin{cases} 10 - \text{x(y)} & \text{if } x(y) < 10 \\ 0 & \text{otherwise} \end{cases}$$

has a value of 4. The x value still takes the **FALSE** branch of $P1$. The value of y is increased by 1 to decide whether positive direction is minimizing the objective function $\Im(y)$ or not. The value of $\Im(y)$ is decreased to 2. This informs that the value of y should be modified in the positive direction. After the value of y is modified by the *step size* which determines the extent of the input modification, then we can get the desired input which minimizes the objective function $\Im(y)$.

The Korel's method also used dynamic data flow analysis to speed up the search process of the test data generation algorithm.

2.3.2 *Test data generation based on genetic algorithm*

In the dynamic approach, most existing techniques solve the function minimization problem with local optimization algorithm [Korel 1990a; Gallagher and Narasimhan 1997], but this suffers from some well-known weaknesses. Local optimization algorithm can fail if a local minimum is encountered. A local minimum occurs when none of the changes of input values that are being considered leads to a decrease in the function value and, yet, the value is not globally minimized (in Figure 2). In other areas where optimization is used, the problem of local minima has led to the development of function minimization methods that pursue a global minimum. One of such examples is genetic algorithm [Gen and Cheng 1997]. The basic genetic algorithm is described in Figure 3. The algorithm is started with a set of candidate solutions (represented by chromosomes) called population. Solutions which are selected to form new solutions (offspring) are selected according to their fitness - the

1. **[Start]** Generate random population of n chromosomes
2. **[Fitness]** Evaluate the fitness $f(x)$ of each chromosome x in the population
3. **[New population]** Create a new population by repeating following steps
   - **[Selection]** Select two chromosomes from a population according to their fitness
   - **[Crossover]** With a crossover rate cross over the selected chromosomes to form new offspring(children).
   - **[Mutation]** Mutate the new offspring at each locus(position in chromosome) which has a mutation rate.
   - **[Accepting]** Place new offspring in a new population
4. **[Replace]** Use new generated population for a further run of algorithm
5. **[Test]** If the end condition is satisfied, stop, and return the best solution
6. **[Loop]** Go to step **2**

Figure 3. Outline of the basic genetic algorithm.

more suitable they are the more chances they have to reproduce. This is motivated by a hope, that the new population will be better than the old one. Some test data generation techniques use the genetic algorithm to solve function minimization [Sthamer 1995; Michael et al. 2001; Pargas et al. 1999]. The implicit parallelism of genetic algorithms allows them to examine many paths at once unlike Korel's method, which must concentrate on a single path. It means that a population in one iteration of the genetic algorithm can examine several paths while several iterations to generate test data for several paths are needed in Korel's method.

The example of test data generation by the genetic search is based on the following piece of code:

```
P1: if (a > 100)
S1:     write("hello world\n");
```

The variable a is the program input. Suppose the test requirement is simply to exercise the **TRUE** branch of the predicate $P1$, causing "hello world" to be printed. The objective function $\Im(a)$ will be $100 - a$ based on the value of variable a.

When the genetic algorithm is invoked, it begins by generating an initial population of test inputs; each input will be treated as one individual by the genetic algorithm. For this example, suppose that four individuals are generated with the values −112, 49, 91, and 94, respectively. The value returned during the four test executions is used to determine the fitness of each of the four inputs. A smaller number indicates that the test input is closer to satisfying the criterion, but, in genetic search, larger numbers have traditionally been associated with greater fitness. Therefore, the fitness of each input is obtained by taking the inverse of the heuristic value returned when the target program is executed. For the inputs −112, 49, 91, and 94, the instrumented program returns 212, 51, 9, and 6, respectively, to the test data generator. Therefore, the respective fitnesses of the four inputs are 1/212, 1/51, 1/9, and 1/6. Once the fitness of each test input has been evaluated, the reproduction phase of the genetic algorithm begins. Two inputs are selected for reproduction according to their fitness. This is accomplished by normalizing the fitness values and using them as the respective probabilities of choosing each input as a parent during reproduction. The probabilities of selecting −112, 49, 91, and 94 are thus 0.015, 0.065, 0.368, and 0.552, respectively. Suppose the two parents selected are 91 and 94, the two inputs with the largest probabilities. Reproduction is accomplished by representing each number in binary form, 01011011 and 01100000, respectively. A crossover point is selected at

random (suppose it is 3), and crossover is performed by exchanging the first three bits of the two parents look like this (| is the crossover point):

Parent 1 : *010* | *11011*
Parent 2 : 011 | 00000
Offspring 1 : *010* | 00000
Offspring 2 : 011 | *11011*

The resulting offspring are 01000000 and 01111011, or 64 and 123.

Since reproduction continues until the number of offspring is the same as the original size of the population, two more inputs are selected as parents. Suppose they are −112 and 91 and the crossover point is 5. If the inputs are converted to bit-strings using 8- bit in two's complement representation, then they are 10010000 and 01011011, respectively. Crossover produces offsprings 10010011 and 01011000, or −109 and 88. In summary, the second generation consists of the individuals 64, 123, −109, and 88. When the program is executed on these inputs, it is found that one of them satisfies the test requirement, so the test generation process is complete (for that requirement). If none of the test inputs had met the requirement, the reproduction phase would have begun again using the four newly generated inputs as parents. The cycle would have been repeated until the test requirement was satisfied or until the GA halted due to insufficient progress.

2.4 Hybrid Approaches

A hybrid of the two forms can be found in two articles by Gupta et al. [Gupta et al. 1998; 2000]. This hybrid approach does not require as many executions to find an appropriate input. This method attempts to overcome the limitations of symbolic execution such as handling arrays and pointers - index of array can be known during execution - by execution of the program when those structures are used. This approach is based on a relaxation technique [Scheid 1968] for iteratively refining an arbitrarily chosen input. So we call this technique iterative relaxation method (IRM)-based method [Gupta et al. 1998]. The relaxation technique is used in numerical analysis to improve upon an approximate solution of an equation representing the predicate function derived from the conditional predicate expression.

This approach can be explained with the following example program below.

```
S1: read(x,y);
P1: if (x+y >= 10) {
S2:     x = x+10;
S3:     y = y+1;
    }
```

Suppose we are trying to find an input to traverse the path $P < S1, P1, S2, S3 >$. The predicate function $\Im$ of the predicate $P1$ is "$x + y - 10$". The predicate slice of the predicate $P1$ and the given path is $S1$. Then the linear function of the predicate function $\Im$ can be derived in the form "$f(x, y) = ax + by + c$". We approximate the derivatives of the predicate function by its divided differences.

To compute a at the initial input $I_0\,(x_0, y_0)$, we execute the predicate slice at I_0. and we choose $\Delta x = 1$, for a unit increment in the input variable x. Then we compute the

divided difference :

$$\frac{\Im(x_0+\Delta x,\ y_0)-\Im(x_0,\ y_0)}{\Delta x}=a$$

Similarly, we compute the value of b.

$$\frac{\Im(x_0,\ y_0+\Delta y)-\Im(x_0,\ y_0)}{\Delta y}=b$$

We compute the value of c by solving for c from the equation

$$ax_0 + by_0 + c = \Im(I_0)$$

Thus, we obtain the linear arithmetic representation for $\Im$ at I_0

If a branch condition does not evaluate to the desired outcome, the evaluation of the branch condition also provides us with a value called the predicate residual which is the amount by which the function value must change in order to achieve the desired branch outcome.

$$ax_0 + by_0 + c = r \ (r \text{ is the predicate residual})$$

Now using the linear arithmetic representation and the predicate residual, we derive a linear constraint on the increments for the current input. One such constraint is derived for each branch condition on the path.

$$a\Delta x + b\Delta y >= -r \ (\text{linear constraint})$$

The derived linear constraints by the iterative relaxation method are solved simultaneously using the unified numerical approach (UNA) [Gupta et al. 1999; Gupta et al. 2004] to compute the increments (Δx, Δy) for the current input. A new input ($x_0 + \Delta x$, $y_0 + \Delta y$) is obtained by adding these increments to the current input. This input value can be test data that satisfy the target test requirement. UNA came from the least squared error solution and it can be stuck in a local minimum during searching for the solution.

3. EVALUATION OF TEST DATA GENERATION TECHNIQUES

In this section, we discuss on how to evaluate representative samples of existing test data generation techniques and compare them systematically according to evaluation criteria through experimentation. Taking the random test data generation as the baseline for comparison, we selected representative techniques for each classied approach. The iterative relaxation method based technique (we call it the IRM-based method) by Gupta et al. [Gupta et al. 1998; 1999; 2000; 2004] represents the hybrid approach. The Korel's method [Korel 1990b; 1990a], and the GA based-method [Michael et al. 2001; Sthamer 1995; Andreou et al. 2007] are chosen to represent the dynamic approach. As test coverage criteria, we chose the branch coverage as a testing criterion because it is commonly regarded as the minimum testing requirements. The branch coverage is that the predicate should be true for at least one input and false for at least one input.

3.1 The Experimental Framework

Figure 4 sketches the procedure of experiments where the boxes representing major

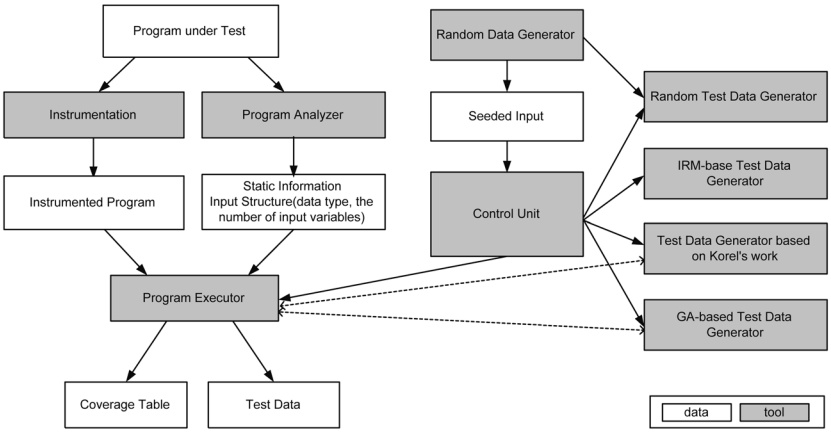

Figure 4. Overall structure for experiments.

tools are shaded.

The control unit is responsible for setting up the environment for experiments; taking the information of the program under test, randomly generating data which are consumed as a seed input for each test data generator except random test data generator and executing a test data generator with the seed input. The search for test data may depend on the quality of the initial input data. Since the initial data to be used for the search are generated randomly, some branches may be reached coincidently. Therefore, a seed input, a random data generated by the random data generator, is supplied to all test data generators for the fair comparison of the test data generation methods. Then, the program under test is executed with this seed input. The covered branch with that seed input is recorded in the coverage table [Michael et al. 2001].

The programming language JAVA (JDK 1.3.1) under the JBuilder 6.0 environment was used to implement each test data generators. The experiments performed on a personal computer with Intel Pentium III 1.01 GHz and 512MB of RAM. We had two choices to implement the test data generators: path-oriented and goaloriented approaches [Edvardsson 1999]. Since we need to create a test set which satisfies branch coverage criteria, we chose the goal-oriented approach. wThe IRM-based method was presented in [Gupta et al. 1998] and the algorithm for branch coverage was proposed in [Gupta et al. 2000]. We do not implement the whole algorithm of the IRM-based method (deriving reachability graph and G" graph, computation of predicate slices, identifying the input dependency sets and path switching). We just provide manually the static information required by the IRM-based method because it takes longer time to implement of the algorithm to extract static information from a program than to implement four test data generators which are for our original purpose of this paper. Instead, we have automated test data generation process and UNA algorithm [Gupta et al. 1999; Gupta et al. 2004]. The Korel's method which was implemented based on papers [Korel 1990b; 1990a] was also implemented as a goal-oriented approach. The GA-based method is based on the algorithms reported

in the literature [Michael et al. 2001]. Parameters and operators in the GA are taken from the experimental results of Sthamer's work [Sthamer 1995].

All of test data generators borrow their overall test data generation algorithms from Michael's work [Michael et al. 2001]. Namely, all test data generators use the coverage table and record a test input which covers a new branch whether or not that test input satisfies the target branches during the search process.

3.2 Evaluation Criteria

We discuss on the three categories which we take into account for evaluating test data generation techniques. Throughout this section, P is a program under test and C is a selected structural test coverage criteria. T is a set of test data which satisfies 100% of test requirements of C for a P. M is a test data generation method and T' is a set of test data found by M for a P and C.

3.2.1 *Performance*

The performance of a test data generation method should be viewed in two different aspects. The first aspect of the performance is the **effectiveness** of a test data generator; the fraction of test requirements covered by T'. Therefore, if we choose the branch coverage criteria, we can get the coverage rate by computing $(n'/n) \cdot 100$ where n' is the number of branches which are covered by T' and n is the number of all branches in the control flow of P except the unreachable branches; if there is no input which traverses the branch, that branch is unreachable. The unreachable branches occur when a programmer make logical errors in a program. If the test data generator can find the test set which satisfies 100% coverage rate, the test data generator is the most effective.

In the other aspect, we measure the **effciency** of a test data generation method in terms of its space and time requirements. **Space effciency** is affected by the amount of information stored during the process of test data generation. In general, plenty of space is required with the static approach, especially the symbolic execution-based method [Gallagher and Narasimhan 1997]. The dynamic approach requires relatively less space because it does not demand the static analysis information.

As far as time is concerned, static approach is more economical than the dynamic approach. While the search process requires a lot of iterations in the dynamic approach, the set of algebraic expressions can be solved all at once in static approach. However, this problem was proved to be NP-complete [Lapierre et al. 1999b]. Hence, it requires a complex computation to obtain a solution from the set of algebraic expressions [Lapierre et al. 1999a].

Implementation of test data generation algorithms and the data structures which are used in test data generators are not optimized. Therefore, we take the different factors when evaluating the **time effciency** of test data generation instead of calendar time or system clock. Since the dynamic approach makes iterative improvements to a test input while trying to make that input satisfy the chosen requirements, we cannot predict exactly how many iterations it needs. However, the most significant computational cost is that of executing the program under test in iteration. Therefore, the number of executions of the program under test is measured to evaluate the time

Table I. Mutant operators to be applied on the example programs.

Mutant operator	Description	Example
SSDL	Statement deletion	$x-=y; \rightarrow ;$
OAAN	Arithmetic operator mutation	$a+b \rightarrow a*b$
ORRN	Relational operator mutation	$a<b \rightarrow a<=b$
OLNG	Logical negation	$x\ op\ y \rightarrow !(x\ op\ y)$
VTWD	Twiddle mutation	$p=a+b \rightarrow p=a+b+1$

effciency [Michael et al. 2001].

Static approach and hybrid approach (IRM-based method) must analyze the program under test and derive the algebraic expressions, then compute the algebraic expressions to find out the desired test data. So we decided to measure the computation time required to analyze the program under test and to solve the algebraic expressions. The IRM-based method also executes the program. However, we take the number of computation residuals by UNA as the measure of time effciency of the IRM-based method, because it is the major portion of the test data generation time as the results of the experiments in [Gupta et al. 1999; 2004].

In summary, the performance of test data generation algorithms is determined by how fast the test data generator can generate the set of adequate test data which has a high coverage rate for a given coverage criteria.

3.2.2 *Quality of test data*

The quality of test data is related to how many faults are detected by T'. If a set of test data T'_1 can reveal more faults in P than the other set of test data T'_2 for a given M and C, we can say T'_1 has a better quality than T'_2. Thus, we measure the quality of test data generated by each test data generator by seeding errors into P. We have introduced selective mutation testing technique to measure the quality of the generated test data [Offutt et al. 1996].

Mutation testing is a fault-based testing technique that measures the effectiveness of test cases. Mutation testing is based on the assumption that a program will be well tested if a majority of simple faults are detected and removed. Simple faults are introduced into the program by creating a test of faulty versions, called mutants. These mutants are created from the original program by applying mutation operators, which describe syntactic changes to the programming language. Test cases are used to execute these mutants with the goal of causing each mutant to produce incorrect output. A test case that distinguishes the program from one or more mutants is considered effective at finding faults in the program. A mutant which produces an incorrect output when executed with a test case is said to be killed. A test case set that can kill all the mutants of a program is called a mutation adequate set.

The *mutation score* for a set of test data is the percentage of non-equivalent mutants killed by that data. A test data set is called mutation-adequate if its mutation score is 100%. We regard that the mutation-adequate test data set has as a good quality. In order to produce mutants of the original program P, we applied several mutant operators to the program under test. The mutant operator is a rule

according to which P is changed. For example, suppose that a mutant operator generates two mutants of P by replacing a use of x by $x+1$ and $x-1$. When applied to a program containing the assignment statement $z = x + y$, this mutant operator will generate two mutants of P, one obtained by replacing this assignment by $z = x + 1 + y$ and the other by replacing this assignment by $z = x - 1 + y$. Please refer to [Offutt et al. 1996] for detail description of mutant operators. We use mutant operators in Table I for mutation testing.

3.2.3 *Generality*

The generality of a test data generation method indicates its ability to function in a wide and practical range of situations. Ideally, the test data generation method should function in the presence of arbitrarily complex programs. The less the test data generation method is restricted by language constructs and target languages in which the program is written, the more generally applicable the test data generation method is.

The test data generation method should work on the complex program to be used in practice. We need to examine the coverage rate in P by T' according to the complexity of each program. If a test data generator covers all target branches on the complex program, we can say that the test data generator is generally applicable.

3.3 Selection of Programs

Before preparation for our experiments, we selected some programs on which we will apply those four test data generators. We selected target programs according to several features (loop, array, non-linear predicates and module calls), input data type (integer and float point number) and the complexity of a program (McCabe's cyclomatic complexity metric [McCabe 1976] and *compl*(*nesting, condition*) [Michael et al. 2001]). McCabe's cyclomatic complexity metric is used to measure the complexity of a program. It directly measures the number of linearly independent paths through a program's source code. The cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to the statements of a program, and a directed edge connects two nodes if the second statement might be executed immediately after the first statement.

The cyclomatic complexity of a structured program is defined as:

$$M = E - N + 2P$$

where M is the cyclomatic complexity, E, the number of edges of the graph, N, the number of nodes of the graph, P, the number of connected components. Next, *compl*(*nesting*, *condition*) was suggested to concentrate two characteristics of a program: 1) how deeply conditional clauses were nested (this is called the nesting complexity), and 2) the number of Boolean conditions in each decision (which is called the condition complexity). For example, a program with no nested conditional clauses (*nesting complexity* = 0) would look like this:

```
if ( (i <= 0) || (j <= 0) || (k <= 0) ) {
    return 4;
}
```

in that if statements are not nested inside other if statements. The nature of the conditional expressions in each conditional clause is controlled by the second parameter (the condition complexity). The decision $((i <= 0)||(j <= 0)||(k <= 0))$ above ranks as a 3 on this scale because it contains three conditions. These two parameters is used to characterize the complexity of our selected programs. The expression $compl(x, y)$ is used as shorthand for nesting complexity x and condition complexity y.

The features of the programs selected for our experiment are briey described below.

triangle. The *triangle* program accepts as input three integers, the sides of a triangle. The output of the program is the type of the triangle determined by the three sides: Equilateral, Isosceles, Scalene, or NotATriangle. This *triangle* program consists of only 'if' statements. Values of input variables are not modied in the *triangle* program. The predicates in this program are a linear function of input variables and some predicates are dependent only on the local variable, which prohibits the test data generation methods from finding the test data which covers that predicate.

remainder. The *remainder* program was chosen to compare how the test data generation methods can handle the loop structure in the program. The remainder program accepts two input variables a and b of type integer and returns the remainder as an output variable *rem* (of integer type) when a(dividend) is divided by b(divisor). The remainder is evaluated by repeatedly subtracting b from a within a loop until the results becomes less than b. Six cases arise from all possible combinations of zero, positive and negative values for a and b. Two out of these six cases are when either a or b is zero, in which case a remainder of 0 is returned. The other four cases arise from the combination of a $a \neq 0$ and $b \neq 0$ being positive and negative, for which the remainder is evaluated. These are treated separately by using the relational operator '<' or '>'. Therefore, the *remainder* program has four different loop conditions.

expint. The exponential program has branch predicates computing linear and nonlinear functions of input. It includes integer input as well as floating point input and three loop constructs.

calcRate. This example is more typical of commercial computing. It contains a mixture of computation and decision making. The *calcRate* program computes its basic automobile insurance rates using the several rules. This program accepts six input variables of which three input variables are of type integer, two are of type Boolean, and one is of type real. These input variables are used to compute the insurance rates and are not modified during the execution.

calcRiseSet. This program is used in astronomy. This computes the times of sunrise and sunset for a given date. The *calcRiseSet* program has a mix of three integers and four floating point inputs and six function calls. Functions which are called by this program are in the java libraries or user defined.

3.4 Results from Experiments

The experiments were performed with five programs to compare four test data

Table II. The complexity of the example programs.

Programs	*triangle*	*remainder*	*expint*	*calcRate*	*calcRiseSet*
Cyclomatic complexity	8	10	15	30	41
Nesting complexity	compl(4, 3)	compl(4, 1)	compl(7, 3)	compl(4, 5)	compl(5, 2)
Input variables #	3	2	2	6	7

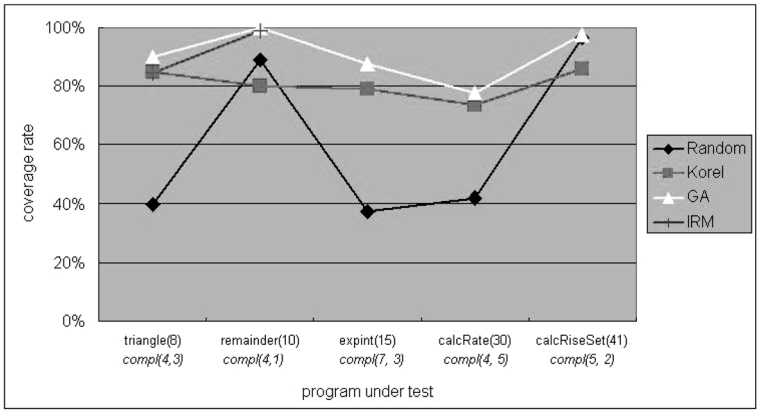

Fig. 5. A graph comparing the effectiveness of four test data generation methods on the five programs with different complexities.

generators. We should execute manually the some steps, extracting static information of a program, of the IRM-based test data generation process because the IRM-based method was not fully implemented as described in previous subsection 3.1. Because of difficulties of executing manually, The IRM-based method was performed on only the *triangle* and the *remainder* program.

In this section, we use three evaluation criteria to evaluate the representative test data generation techniques. First, we investigate the performance of each test data generation technique with the results of experiments. Analysis of experimental results involves the quality of test data of each test data generation obtained by analyzing the outcomes from error seeding testing with some mutant operators. Finally, we present how well each test data generator can treat the complex program and can be generally applicable.

3.4.1 *Examination from aspect of performance*

We analyze the effectiveness of a test data generator by examining the percentage of the covered branches of each method. In figures 5, the graph shows the coverage rate by each test data generator in each example program. The coverage rate of the random test data generator is extremely low with about 40% coverage rate in three programs (*triangle*, *expint* and *calcRate*), while the random test data generator has higher coverage rate than that of the Korel's method in two programs (*remainder* and *calcRiseSet*). This results from the distribution of the data which traverses the target branches. The probability of selecting an input that traverses the target

Table III. The time efficiency and the space efficiency of the test data generations (program under test information is less, the space efficiency is better because a test data generator does not store information of a program under test).

Methods	random	IRM	Korel	GA
average coverage rate (%) for all programs under test	61	90	81	91
execution times (iteration count)	20000	779	367	7991
program under test information	nothing	control flow graph, reachability graph, predicate slices, a derived constraints	data flow analysis information	no static analysis information

branches uncovered by the random test data generator is lower than the others. This is the weakness of the random test data generation as discussed in previous section 2.1. But, most data that traverse the branches in the *remainder* and the *calcRiseSet* program are distributed widely. In result, the random test data generator can have high coverage rate in two programs owing to the high probability of selecting the data that traverse the target branches. On the other hand, the Korel's method was often stuck in the local minima during the search process in those two programs. Also the values of the variables in the algorithm affected the coverage rate. For example, the value of the *step size* which determines the extent of the input modification to meet the test requirement is not fixed but given by user. The Korel's method covered an average of 73–85% of branches of each program. The coverage rate of the IRM-based method is lower than the GA-based method. The IRM-based method can be stuck in a local minimum while computing the residuals by the UNA [Gupta et al. 1999; Gupta et al. 2004]. The increments (Δx, Δy) from UNA computation for next input data ($x_0 + \Delta x$, $y_0 + \Delta y$) depend on an initial input data generated randomly; some random initial input data leads to the value which will cover the target branch, others leads to a local minimum which will not cover the target branch. So the IRM-based test data generator did not work properly. However, its coverage rate is almost equal to the Korel's method in the *triangle* program. In the *remainder* program, the IRM-based method has higher coverage rate than the Korel's method. The GA-based method achieved the best coverage rate for all programs.

Table III shows how many iterations each method carries out to achieve its average coverage rate for all five programs under test except IRM-based method (IRM-base method generated test data for only two programs described before). The time efficiency of the test data generators, on the whole, the GA-based method requires a lot of executions of the program under test than the Korel's method. This is understandable because the GA-based method uses the global optimization algorithm and the Korel's method uses the local optimization algorithm. In general, the local optimization algorithm is faster than the global optimization algorithm because the global optimization needs more search for global minimum. The IRM-

based method is faster than the Korel's method, although the iteration count is bigger than the Korel's method; the computation of algebraic expressions is faster than the program execution. All of the test data generators expend a lot of search time or computing time until the test data generators fail. However, we cannot handle this problem because the test data generation problem is an NP problem; we do not know there is the test data which traverses the target branches. Therefore, we need to give a proper limit to each test data generator.

As for space efficiency, the random test date generator is the best space efficient method because it does not require any program information except the number of input variables and types. The GA-based method is the second-best space efficient method because the GA-based method does not require the static analysis information. However, if the program under test has many input variables, the length of chromosomes is long. The space and time efficiency of the GA-based method decrease as the chromosomes become longer. The Korel's method uses data flow analysis information even if data flow is analyzed dynamically. Therefore, the Korel's method needs more space to store that information than the GA-based method. The IRM-based method uses a lot of static analysis information such as control flow graph, reachability graph and predicate slices, and a derived constraints of system (algebraic expressions). Therefore, the IRM-based method requires more space to store the static information than the Korel's method.

3.4.2 *Examination from aspect of quality of test data*

The results from the selective mutation testing are showed in Table IV. The mutation scores do not give us which method generates the better quality of test data directly. So we deduced which method generates the better quality of test data from the relative mutation scores.

There are two reasons why some mutants are not killed. Some mutants are live if the set of test data does not cover all branches, and some seeded errors are located in the uncovered branches. The coverage rate almost matches the quality of test data in this case, in other words, the test data generation algorithm which achieves a high coverage rate gets a set of good quality of test data. The second reason is that most mutants generated by the relational operator mutation (in Table I) were not killed. This mutation operator changed the relational operator, for example, the code '>' into '>='. If we do not have test data in the boundary of input domain, those mutants will not be killed. The test data generator must find out the boundary values to kill those mutants. However, any of test data generators in our study does not focus on the boundary values. Therefore, if the boundary values are required, the test data generators should be modified as such. The Korel's method has a larger mutation score than the other methods with the *remainder*, and the *expint* programs, even though the covered percentage is lower. It is because that the Korel's method found out more boundary values than the other methods. As explained in subsection 2.3.1, the Korel's method minimizes the objective function for local optimization. That is, the input values generated the Korel's method have high chance to be boundary values which minimize the objective function. The GA-based method for global optimization has less chance to find out boundary values for the input values because

Table IV. Relative mutation scores and an average relative mutation score of each method which was computed by *mutation score/coverage rate*.

Methods	random	Korel	GA	IRM
triangle	94% (90.4%/96%)	84% (78.3%/93%)	96% (90.4%/94%)	66% (56.5%/85%)
remainder	73% (72.5%/100%)	99% (82.5%/83%)	69% (69.2%/100%)	75% (75.0%/100%)
expint	91% (72.5%/80%)	90% (75.0%/83%)	79% (69.2%/88%)	
average	86%	91%	81%	71%

it expands the search domain of the input values to find out global optimum. The input values generated by the IRM-based method are different from the boundary values by the increments to be added to the previous input in explained in subsection 2.4. So it is difficult for the IRM-based method to find out boundary values. In result, the test set of the Korel's method killed more relational operator mutated mutants and the average relative mutation score of the Korel's method is highest.

It is useful to test the program with several sets of test data, though a test data generator cannot generate the set of test data which does not achieve 100% coverage for the given coverage criteria. Since one set of test data may contain some test inputs which cover some target branches not covered by other sets of test data. However, the test data generated by the IRM-based method for several tries are almost same although the initial random data has different values. Thereby, the number of killed mutants by several sets of test data and the number of killed mutants by one set of test data are the same. The IRM-based method is not suitable in a case where multiple test sets are needed.

3.4.3 *Examination from aspects of generality*

For capability of handling program constructs of each test data generator, briey, the most programming constructs are transparent to the GA-based method and it needs only the information for calculating the objective function. So the GA-based method is general. The Korel's method is based on the dynamic approach, but it uses static analysis information - dynamic data flow analysis to improve the test data generator. The IRM-based method uses the static analysis information mainly, aided by the dynamic information to handle the array constructs. So the IRM-based method is sensitive to the program constructs.

Next, we examined the coverage rate for each method as the complexity of the program under test increases. This analysis examines how well the test data generators performs as the complexity of the program increases. Figure 5 shows which test data generation method performs better with the complex program. The GA-based test data generation performed the best overall.

It is interesting to note that the coverage rate did not just decrease as the cyclomatic complexity of the program increases. Rather, the coverage rate is related to the condition complexity y of $compl(x, y)$. All test data generator gave good coverage rate on the *remainder* program and the *calcRiseSet* program which have the condition complexity of 1 and 2 each. On the other hand, all generators except Korel's method performed worse on the other programs (the condition complexity of

Table V. Overall scores to rank four test data generation methods for all suggested criteria. *Time* stands for time efficiency, *Space*, space efficiency, *Con*, generality for programming constructs, and *Com*, generality to complexity of a program.

Methods	random	GA	Korel	IRM
Effectiveness(1)	1	4	2	3
Efficiency(1) (*Time* * 0:8 + *Space* * 0:2)	2.0 (1 * 0:8 + 4 * 0:2)	2.5 (3 * 0:8 + 2 * 0:2)	3.0 (3 * 0:8 + 2 * 0:2)	3.5 (4 * 0:8 + 1 * 0:2)
Quality of Test Data(1)	3	2	4	1
Generality(1) (*Con* * 0:5 + *Com* * 0:5)	2.5 (4 * 0:5 + 1 * 0:5)	3.5 (3 * 0:5 + 4 * 0:5)	2.5 (2 * 0:5 + 3 * 0:5)	1.5 (1 * 0:5 + 2 * 0:5)
average	2.1	3.0	2.9	1.5

3 and 5) than the above two programs. In most cases, the coverage rate decreases as the condition complexity of the program increase in Figure 5.

The results may be due to the distribution of the solution (the adequate test data that traverses the target branches). The number of Boolean conditions in each decision (the condition complexity) limits the distribution of the solution. In the case of the *remainder* program, the input values that traverse the predicate 'a > 0' are all positive integers. The input values which satisfy the target branches are distributed over the wide range of input domain. So, the test data generators easily found the test data that cover the predicate. For the predicate '$i == j$ && $i + j > k$' in the *triangle* program, input values are distributed in the range of the input domain that satisfies the predicate expression. The input values are distributed on the narrow range of the input domain. This narrow distribution of the input domain prohibits the test data generators from finding out the test data. We notice that the distribution of solutions for the target branches inuences the performance of the test data generators. Namely, the higher the condition complexity is, the narrower the distribution of solution is.

3.4.4 *Further discussion*

Finally, we used the simple and heuristic way to rank four test data generation method as a whole. We gave a grade from 4 to 1 to each test data generation method according to the ranking of the method in a specific criterion; grade 4 means that the method is the best in that criterion and grade 1 means that the method is the worst. The ratio of each criterion is same as there is no study which criterion is more important. The **effectiveness** of the **performance** criterion has the same weight with other criteria because it is the basic requirement of test data generation method. **Time efficiency** is more important than **space efficiency** in the **performance** criterion because space used by a program recently is not as important as before, i.e., memory is cheaper these days. Therefore, we gave the **time efficiency** the weight of 0.8; the **space efficiency**, 0.2. The programming constructs and the complexity of a program in the **generality** criterion have the same weight of 0.5.

We ranked four test data generation methods according to the average coverage

rate in Table III when evaluate the effectiveness. The time and space efficiency was evaluated according to our analysis and Table III. The test data generator methods scored according to our analysis in Section 3.4.3 and Figure 5. The average score in Table V shows that the GA-based method is the best among the methods compared in this paper.

There are some problems to be discussed on the experiments. In the *triangle* program, the common problems of all test data generators were discovered through the experiments. If a conditional predicate was only dependent on some variables which were not data-dependent on the input variables, all test data generators failed to discover the test data covered that branch. Some predicates in the *triangle* program have the local variable *tri* which is not data-dependent on the input variables. The value of *tri* is determined by the predicate-use of input variables. The GA-based method uses the predicate information to compute the fitness function, so it is difficult to identify the chromosomes with better fitness values. The Korel's method use the predicate information to compute the objective function values which guide the direction of search. In addition, the Korel's method cannot handle the target branches in this case since the Korel's method only considers the input variables on which variables of the predicate expression are dependent. The IRM-based method computes the system of constraints to find the test data. The system of constraints consists of the linear functions of input variables, which is derived from the predicate functions in the given path. If the predicate expression is '$x + y > 10$', the predicate function is '$x + y - 10$' in the IRM-based method. Then, the IRM-based method computes the linear function of the predicate function with input variables.

The linear function is derived in the form '$f(x, y) = ax + by + c$' where x and y are input variables. However, if the variables in the predicate function are not dependent on the input variables, the IRM-based method cannot derive the linear function from that predicate expression. Since the variable *tri* in the predicate is not dependent on any input variables, the IRM-based method cannot derive the linear function from that predicate.

The GA-based test data generator slowed down when it carried out on the program with loops. The speed of the GA-based depends on the number of iterations. The length of chromosomes, the population size, affects the speed of the GA-based method.

4. IMPROVEMENT OF TIME EFFICIENCY OF THE GA-BASED TEST DATA GENERATION

The GA-based method turned out to perform the best overall except time efficiency according to our experiments described in previous section. In this section, we introduce two ways to improve the speed of the GA-based method. First, the GA-based test data generation algorithm with static analysis information is explained. Second, we can combine the local optimization algorithm and the GA. Then, the experimental results of the standard GA-based method and two modified versions are to be discussed.

4.1 The First Version with Static Analysis Information

We added some features to the standard GA-based technique to relax the limitation

related to the time efficiency. We try to solve two problems in the dynamic approach. One problem is that the speed of test data generator becomes slower by executing the program under test. Especially, the GA-based method needs many more program executions than the test data generation technique using local optimization algorithm (see Table III). Therefore, we try to speed up the test data generator by reducing the number of the program executions. To cure the other problem that a great number of iterations are carried out before a suitable input is found, the number of iterations can be reduced by using the information on the path which the program should traverse to cover a target branch.

The dynamic approach does not use static analysis information such as control flow, control dependency, and data dependency. A test data generator based on the dynamic approach searches test data using the dynamic information by instrumentation without detailed implementation information. Therefore, the dynamic approach requires relatively many iterations before desired solution can be obtained compared to the static approach or the hybrid approach using static analysis information; according to one of our experimental results, the GA test data generator iterates 2,223 times while the IRM-base method iterates 39 times to achieve 80% coverage for *triangle* program. If the search procedure is provided with useful static information to find out test data, there are possibilities to speed up the search process. The Korel's method improved the search performance of the test data generation through dynamic data ow analysis. TGen [Pargas et al. 1999], the GA-based test data generator used control dependency of the program under test to improve the search performance of the test data generation. We are going to use not only control dependence graph, but also partial data flow analysis. This static analysis information is expected to enable the test data generator to reduce the program executions, to find out the test data of the target branches faster and to detect unreachable test requirements (in our case, unreachable branches). A target branch is considered unreachable if no test input executes the target branch until the GA iterations reach the given limitation in the current GA-based test data generators. Therefore, a test data generator cannot distinguish between unreachable branches and GA-failures (genetic search fails to find out test data although there is an input that traverse the branch). Only if we know a branch is unreachable, we can avoid a lot of iterations of the GA. The algorithm we present is based on the infeasible path detection algorithm [Bodik et al. 1997].

First, we intend to reduce the number of the program executions the following way. If a predicate P has an expression which consists of only input variables and constants, and the input variables are not modified on subpaths from entry node of control flow graph to that predicate node, then the fitness function can be computed without executing the program under test. For example, consider the next predicate `if` condition where a, b, and c are input variables,

```
if ( a + b > c )
```

If there is no assignment that modifies the input variables a, b, and c from entry of the program to this `if` condition, the fitness function of this predicate can be computed directly without executing the program. If a fitness function is computed by subtracting

the value of the right operand from the value of the left operand, the value of fitness can be computed by '$a + b - c$'. Note that there are some conditions when the predicate is nested. Suppose the nested predicate P is control dependent on a predicate P'. If the predicate P' consists of the other local variables, it is unknown whether the target predicate P is reached by the data to be evaluated until the program is executed. Only if the predicate P' consists of the input variables and constants and is not in a loop statement, the fitness function of the target predicate P can be computed without executing the program.

Second, we intend to avoid a lot of iterations of the GA to find out test input that traverses the target branch. We utilize the control dependence graph of the program under test and data flow information. The control dependence graph includes information that corresponds to the following assertions.

—**Assertion 1** : a constant assignment to a variable may imply a particular direction of the conditional.

—**Assertion 2** : a prior conditional branch may subsume the branch predicate P.

We call the predicate of the target branch as the target predicate from now on. How to extract the information for the calculation of fitness function will be described. In general, the fitness function can be computed by calculating the difference between two operands in the predicate expression of the target branch. Assume that '$(x > 0)$' is a predicate expression. A difference between the value of x and 0 is used to compute the fitness of a chromosome. We utilize not only the value of expression in target predicate but also the additional information of other predicates to calculate the fitness. Two steps will be performed to determine which predicates in addition to the target predicate can be used to calculate the fitness.

In the first step, we check whether the target branch is reachable or not by *Assertion 2*. Let's consider next example :

```
P0 : if (x>10) {
P1 :    if (x==0) ...
     }
```

If the target branch is the true branch of $P1$, $P1$ is control dependent on $P0$. The predicate expression of $P0$, $x > 10$, subsumes the target predicate $P1$. According to *Assertion 2*, we can determine whether the target branch is reachable or not. The true branch of $P0$ generates the assertion $x > 10$, and then the assertion $x > 10$ determines that the predicate $P1$ always takes the false branch. Therefore, the target branch is unreachable. If it is reachable, then the previous conditional branch is included into information to evaluate the fitness of chromosomes.

In the second step, given a target branch B with the predicate expression P (e.g., $x > 0$), we check if the outcome of $(x > 0)$ is true or not in turn at the nodes of control flow graph backward. This query can be answered by assertions. Whenever the query is answered to *true* or *false* in a node with *Assertion 1*, we find the predicate on which the target branch is dependent. If the query's answer is *true* and the node is true-dependent on that predicate, we add a true branch of that predicate to the information to compute the fitness function. If the node is false-dependent on that

predicate, we add a false branch of that predicate to the information. It is because that the branch must be traversed by the candidate solutions. If the query's answer is *false* and the node is true-dependent on that predicate, we include a false branch. If a node is false-dependent, we include a true branch, because the branch must not be traversed by the input values to cover the target branch.

We evaluate the chromosomes's fitness using predicate information included so far. Next, the genetic algorithm is carried out.

4.1.1 *Fitness function for preferred paths*

If branch information used for fitness evaluation is determined for the target branch, we calculate the fitness of chromosomes. The fitness function as follows.

$$\Im = \sum_{each\ predicate} \begin{cases} \frac{1}{\delta+f} & \text{if a prior branch is reached, but not covered(1)} \\ \frac{1}{c} & \text{if a prior branch is covered by the chromosome(2)} \\ \frac{1}{\delta+f} * d & \text{if a branch is target(3)} \end{cases} \quad (1)$$

where *each predicate* means the predicate information included to compute the fitness, δ, the minimum value, f, a basic fitness function, c, the number of the predicates which are used in fitness evaluation, and d, a large value. In genetic search, larger numbers have traditionally been associated with greater fitness. Therefore, the fitness is obtained by taking the inverse of the heuristic value returned when the target program is executed. So, we adopted δ to avoid zero-numerator problems when we take the inverse of the value computed by the basic fitness function. If the test input (chromosome) reaches the specified prior branch, but did not cover, we calculate the basic fitness(1). This fitness value is to guide the chromosomes which do not reach the target branch to the target branch. If the test input covers the specified prior branch, we add $1/c$ to the fitness value(2). This value provides the chromosome with a good fitness value. As a result, the GA selects the chromosomes which reach the target branch. If the predicate is the target predicate, $(1/(\delta+f))*d$ is added to the fitness(3). This value is used to cover the target branch. That is, the fitness value becomes larger. In the case of the target predicate, the reason that we multiply it by d is to differentiate the branch that is to be traversed from the fitness value for the target branch. That is, we make the fitness value of the test input (chromosome) that reaches the target predicate larger than that of test input (chromosome) which does not reach the target predicate. We evaluate the chromosomes using the predicate information that is determined by assertions to compute the fitness values. The GA is biased by this fitness function to traverse the

```
initialize P;
while (not termination condition) do
  evaluate P;
  select the best chromosome;
  hill climb the best chromosome;
  generate next P (including crossover and mutation);
end
```

Figure 6. The hybrid genetic algorithm.

preferred path. This fitness function can facilitate the convergence of the GAs.

4.2 The Second Version based on The Hybrid GA

We try to speed up the GA-based method by combining the GA and the hillclimbing algorithm [Gen and Cheng 1997] which is a local optimization algorithm. Because of the complementary properties of genetic algorithms and conventional heuristics, the hybrid GA often outperforms either method operating alone [Gen and Cheng 1997]. Thereby, we expect that the hybrid GA reduces the number of iterations required to find out the test data.

We applied the hill-climbing algorithm to the GA by changing 1 bit of a n-bit chromosome and evaluating the changed chromosome. We just use this operation on the chromosome with the best fitness value. If all chromosomes are hill-climbed, the number of program executions is more required than the standard GA-based method.

The basic algorithm is described in Figure 6.

4.3 Experimental Results

In order to verify the improvement in the time efficiency of two modified versions of the GA-based method, experiments were carried out with two modified versions of the GA-based method. The three program examples which are described in the previous section are used: *triangle*, *expint*, and *calcRate*.

The *calcRate* program has two unreachable branches. While the GA-based method is iterated until the limit of iterations is reached, the first version with static analysis information reported those branches are unreachable through static analysis information. In our experiments, the maximum generation number is 100. Therefore, we saved the cost corresponding to the 100 iterations.

Table VI shows the first version speeds up with the average rate of 74% – 80% than the speed of the standard GA-based method and the second version speeds up with the rate of 7%, 11%, 75% in each program. Two modified versions turned out to be faster than the GA-based method from these results, although the speed up rate of the second version is just 11% and 7% for each *triangle* and *expint* program compared to speed up rate of the first version which is 80% and 74%. The execution time of the first version with static analysis information depends on the program under test. Since all example programs have several predicates which can be evaluated without executing the program under test, the execution time of the first

Table VI. Average speed up rate $(((\frac{1}{t(Vn)} - \frac{1}{t(GA)})/\frac{1}{t(GA)}) \times 100)$, reduced tests count rate and average coverage rate (V1: the first version with static analysis, V2: the second version based on the hybrid GA, GA: the standard GA-based method, t: execution time of a test data generator to reach the same coverage rate, Vn: V1 or V2).

	speed up rate		reduced tests count rate		coverage rate		
Programs	V1	V2	V1	V2	V1	V2	GA
triangle	80%	11%	13%	50%	91%	89.5%	90%
expint	74%	7%	129%	126%	86.67%	91.25%	87.5%

version can be reduced. The speed up of the second version based on the hybrid GA resulted from the reduction of the number of iterations during the search process.

The tests count (table VI) means the number of the chromosomes checked to determine whether a chromosome covers the target branch or not until the new test data is found. The tests count of two modified versions were reduced in the *triangle* program. We notice the tests count of two versions was rather increased about 20% in the *expint* program. For the first version, the effect of the serendipitous coverage [Michael et al. 2001] cannot play a role in the algorithm of the first version. Namely, while the GA-based method records a new input which traverses the new branch even if that branch is not targeted, the first version cannot record such an input because the first version does not execute the program if the target predicate consists of the input variables and constants and does not know whether a certain chromosome (input) traverses a new branch or not. Test data which traverse not the target branch but a new branch formed about 55% of the test set generated by the GA-based method in the *expint* program. Therefore, the first version required more iterations to achieve almost equal coverage rate to the GA-based method. In the case of the second version, the increased tests count is due to the higher coverage rate. The tests count of the second version was rather fewer than the standard GA-based method until the coverage rate reached 87.5%. The equality conditional predicate ('$((n == 0 \| n == 1) \&\& x == 0.0)$') in the *expint* program was covered by only the second version. Since the probability of finding out the data which traverses this predicate is extremely low, more iterations are required to find out the data which traverse that predicate in the second version.

In the GA-based test data generation, there are no previous works with static analysis information except [Pargas et al. 1999]. TGEN is compared only to the random test data generation. The first version with static analysis information can detect the unreachable branches before starting the genetic search and facilitate the convergence of the GA. The experiments show that the first version keeps the coverage rate of the standard GA-based method and speeds up the standard GA-based method. The second version based on the hybrid GA also speeds up the standard GA-based method. In addition, the second version raise the coverage rate of the standard GA-based method because the second version performs better on the predicate with equality condition as compared to the standard GA-based method.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have evaluated the representative test data generation techniques through experiments. We took the random test data generation as a beseline for comparison, and selected the IRM-based method to represent the hybrid approach [Gupta et al. 1998]. The Korel's method [Korel 1990b; 1990a], and the genetic algorithm based method [Michael et al. 2001; Sthamer 1995] represent the dynamic approach. The results of experiments can guide the choice of a test data generation method for practical applications. If a fast test data generator is required, the IRM-based method and the Korel's method is a proper method. However, it should be improved to achieve a high coverage. The work on this issue was performed already [Ferguson and Korel 1995]. If a high quality of the program is required, the GA-

based method can be chosen. But, it should be improved to be faster and not to fail in finding out the test data because it requires a lot of execution of the program under test until the failure is reported. Therefore, two modified versions of the standard GA-based test data generation algorithm have been proposed. We have conducted experiments on the two versions of the GA-based method. As a result, The speed up of the GA-based method is achieved in two modified versions.

For more detailed evaluation of test data generation methods, we need more experiments in the quality of test data and generality criteria. We can acquire more information by performing experiments according to the factors which affect the quality of test data and according to the mutant operators. We need to include the Korel's method with chaining approach [Ferguson and Korel 1995]. Since the chaining approach inuences the effectiveness of the Korel's method. The results of our experiments showed that the generality criterion is related to the distribution of input domain. So we need to analyze the relationship between generality and input domain in the future work. Also we need more experiments with the IRM-based method through automation to be compared more fairly.

REFERENCES

Andreou, A. S., Economides, K. A., and Sofokleous, A. A. 2007. An automatic software test-data generation scheme based on data flow criteria and genetic algorithms. In *Proceedings of the 7th IEEE International Conference on Computer and Information Technology*, 867–872.

Beizer, B. 1990. *Software Testing Techniques(2nd edition)*. Van Nostrand Reinhold.

Bodik, R., Gupta, R., and Soffa, M. L. 1997. Refining data flow information using infeasible paths. In *Proceedings of the Sixth European Software Engineering Conference (ESEC/ FSE 97)*, M. Jazayeri and H. Schauer, Eds. Lecture Notes in Computer Science 1013, Springer–Verlag, 361–377.

Clarke, L. A. 1976. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering, 2*(3):215–222.

Coward, P. D. 1988. Symbolic execution systems-a review. *Software Engineering Journal 3*, 6, 229–239.

Edvardsson, J. 1999. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering in Linkoping,* 21–28.

Ferguson, R. and Korel, B. 1995. Software test data generation using the chaining approach. In *ITC*. IEEE Computer Society, 703–709.

Gallagher, M. J. and Narasimhan, V. L. 1997. ADTEST: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering, 23*(8):473–484.

Gen, M. and Cheng, R. 1997. *Genetic Algorithms and Engineering Design*. John Wiley and Sons, Inc., New York.

Gupta, N., Cho, Y., and Hossain, M. Z. 2004. Experiments with una for solving linear constraints in real variables. In *Proceedings of the 2004 ACM symposium on Applied computing*. ACM, 1013–1020.

Gupta, N., Mathur, A. P., and Soffa, M. L. 1998. Automated test data generation using an iterative relaxation method. In *Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering (FSE-98)*. Software Engineering Notes, *23*(6), ACM Press, 231–244.

Gupta, N., Mathur, A. P., and Soffa, M. L. 1999. UNA based iterative test data generation and its evaluation. In *14th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society Press, 224–234.

Gupta, N., Mathur, A. P., and Soffa, M. L. 2000. Generating test data for branch coverage. In *Proceedings of IEEE International Conference on ASE,* 219–228.

Korel, B. 1990a. Automated software test data generation. *IEEE Transactions on Software Engineering, 16*(8):870–879.

Korel, B. 1990b. A dynamic approach of test data generation. In *Conference on Software Maintenance,* 311–317.

Lapierre, S., Merlo, E., Savard, G., Antoniol, G., Fiutem, R., and Tonella, P. 1999a. Automatic unit test data generation using mixed-integer linear programming and execution trees. In *Proceedings of IEEE International Conference on Software Maintenance.* IEEE Computer Society Press, 189–198.

Lapierre, S., Merlo, E., Savard, G., Antoniol, G., Fiutem, R., and Tonella, P. 1999b. Automatic unit test data generation using mixed-integer linear programming and execution trees. In *Proceedings of IEEE International Conference on Software Maintenance(ICSM '99).*

McCabe, T. J. 1976. A complexity measure. *IEEE Transactions on Software Engineering, 2*(4): 308–320.

Meudec, C. 2001. ATGen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability, 11*(2):81–96.

Michael, C. C., McGraw, G., and Schatz, M. 2001. Generating software test data by evolution. *IEEE Trans. Software Eng., 27*(12):1085–1110.

Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., and Zapf, C. 1996. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology, 5*(2):99–118.

Pargas, R. P., Harrold, M. J., and Peck, R. 1999. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability, 9*(4):263–282.

Scheid, F. 1968. *Numerical Analysis, Schaum's Outline Series.* McGraw-Hill Book Company.

Sthamer, H. H. 1995. The automatic generation of software test data using genetic algorithms. Ph.D. thesis, University of Glamorgan.

Seung-Hee Han received the BS degree in computer science from Yonsei University in 1996, the MS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST) in 2003. She worked as a designer and developer for the TongYang Systems between 1996 and 1999. During the year of 2003, she was as a researcher of Digital Multimedia laboratory in LG Electronic Technology Institute. Since 2004, she has been working as a senior researcher of Network Technology Laboratory, Korea Telecom, Daejeon, Korea. Her current research interests include quality assurance for software and service management at application level for next generation network services.

Yong-Rae Kwon received the BS and MS degrees in physics from Seoul National University in 1969 and 1971 respectively, and a PhD in physics from the University of Pittsburgh in 1978. He taught as an instructor at the Korea Military Academy from 1971 to 1974. He was on the technical staff of the Computer Science Corporation from 1978 through 1983 working on the ground support software systems for NASA s satellite projects. Since 1983, he has been working as a Professor of Software Engineering in the Department of Electrical Engineering and Computer Science at the KAIST. His research interests include verification of real-time parallel software, object-oriented technology for real-time systems, and quality assurance for high-dependable software.