University of Alberta

Post Weld Heat Treatment – PWHT: Temperature-Time Relationships

by

Marcel Zulic ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science

in

Welding Engineering

Department of Chemical and Material Engineering

Edmonton, Alberta

Spring 2004

Your file Votre référence
ISBN: 0-612-96576-7
Our file Notre référence
ISBN: 0-612-96576-7

Abstract

PWHT – Post Weld Heat Treatment is aimed at reducing the residual stresses resulting from manufacturing and welding. This treatment also exerts metallurgical changes influencing the properties of the base metal, weld metal and the heat-affected zone (HAZ).

The thermal cycle of the heat treatment after welding is governed in the codes by the heating rate, the holding temperature, the holding time and the cooling rate. There is a relatively good agreement among Codes and Standards about the maximum tempering temperature from 600^{0}C to 680^{0}C, while there is discordance about the holding time at the same tempering temperature. For medium carbon and low alloy steels, there is little information about PWHT below 600^{0}C.

PWHTs at 550^{0}C and 500^{0}C with different holding times were performed for medium carbon 1043 and low alloy 4140 steels welded with the SMAW process. Hardness measurements of the base metal (BM), heat-affected zone (HAZ) and weld metal (WM) and microstructural examinations were done to correlate the steel properties with different PWHT parameters.

Both steels can be post weld heat treated at temperatures under 600^{0}C.

PWHTs at 550^{0}C/4.0 hours and 500^{0}C/15.0 hours produced similar results to these of recommended PWHTs at 600^{0}C and 600^{0}C. Tempering parameters based on diffusion coefficients are slightly conservative while those based on the Hollomon-Jaffe parameter are overly conservative.

CONTENT

List of Table

List of Figures

1. Software Testing and Test data Generation

1.1. Software testing

1.1.1. Introduction

As long ago as early 70's, it was pointed out that testing for correctness is futile. Testing can only determine the presence of errors not their absence. Even a small program can contain many millions of possible test-input conditions. Consequently, exhaustive testing is impractical. Despite these fundamental limits to software testing, it is still the most commonly used and potent weapon against software errors.

Software testing is an expensive component of software development and maintenance, since some researches indicate that approximately 50% of the software production development cost is spent on software testing. It consumes resources and adds nothing to the product in terms of functionality. Therefore, much effort has been spent on development of automatic software testing tools in order to significantly reduce the cost of software testing, so as to reduce the cost of software development and maintenance. In order to test software, test data have to be generated and some test data are better than others when they are used to find errors. Therefore, a systematic testing system has to differentiate good (suitable) test data from bad test (unsuitable) data, and it should be able to detect good test data if they are generated. A search algorithm is needed to decide where the best test data lie and concentrate its search there. Nowadays testing tools can automatically generate test data, which will satisfy certain criteria, such as branch coverage, path coverage, etc. A testing tool should be general, robust and generate the correct test data corresponding to the testing criteria.

Test data that are appropriate for one program are not necessarily appropriate for another program even if they have the same functionality. Therefore, an adaptive testing tool for the software under test is necessary. Adaptive means that it monitors the effectiveness of the test data to the environment in order to produce new solutions with the attempt to maximize the test effectiveness.

There are basically two software-testing methods: black box testing (sometimes called "functional" or "specification-based") and white box testing (sometimes referred to as "structural" or "code-based" or even "glass-box"), Roper [1994].

1.1.2. Black Box Testing

The internal structure and behavior of the program under test is not considered. The objective is to find out solely when the input-output behavior of the program does not agree with its specification. The strength of black box testing is that tests can be derived early in the development cycle. The software is treated as a black box and its functionality is tested by providing it with various combinations of input test data.

1.1.3. White Box Testing

The internal structure and behavior of the program under test is considered. The structure of the software is examined by execution of the code and test data are derived from the program's logic. This method gives feedback e.g. on coverage of the software.

There are several white-box testing criteria frequently used by software testing tools:

Statement testing:

Generate test data to execute every source language statement in the program at least once. Also referred to as statement coverage.

Branch testing:

Generate test data to exercise the true and false outcomes of every decision. Also referred to as branch coverage or decision coverage. Branch testing resolves the "null else" problem by forcing the true and false outcomes of each branch, even if there is no code associated with these outcomes. Its weakness is in the testing of compound conditions. For example, given the following situation:

```
...
if A==0.5 or (B>0.5 && C==TRUE&&D=='Yes')
   {some code}
else
   {some code}
...
```

The only thing we need to do to traverse the true outcome is to make *A* equal to 0.5, and we don't have to brother about the rest of the condition.

Condition/decision testing:

Generate test data such that all conditions in a decision take on both outcomes (if possible) at least once, and exercise the true and false outcomes of every decision. Also referred to as decision/condition coverage.

Modified condition/decision testing (MC/DC):

Generate test data such that every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions. Also referred to as MC/DC coverage.

For example, consider the following fragment of code:

```
if (A or (B and C))
   {some code}
else
   {some code}
```

MC/DC requires test cases to show that each Boolean operand (A, B and C) can independently affect the outcome of the decision. MC/DC may be achieved with the following set of test inputs (note that there are alternative sets of test inputs, which will also achieve MC/DC):

Table 1.1 A Set of Test Inputs to Achieve MC/DC

Case	A	B	C	Outcome
1	FALSE	FALSE	TRUE	FALSE
2	TRUE	FALSE	TRUE	TRUE
3	FALSE	TRUE	TRUE	TRUE
4	FALSE	TRUE	FALSE	FALSE

In the above example:

- A is shown to independently affect the outcome of the decision condition by case 1 and case 2;
- B is shown to independently affect the outcome of the decision condition by case 1 and case 3;
- C is shown to independently affect the outcome of the decision condition by case 3 and case 4.

To achieve 100% MC/DC requires a minimum of n+l test cases, and a maximum of 2n test cases

Path testing:

Generate test data to cause execution of all paths in the program. Path testing increases the probability of error detection, and a path through software can be described as the conjunction of predicates in relation to the software's input variables. The weakness is that there are a lot of paths (in worse cases, there might have an infinite number of paths), but this is also the strength in that combinations of paths are exercised, which other structural testing methods do not achieve. However, path testing is generally considered impractical because a program with loop statements may have an infinite number of paths. A path is said to be 'feasible', when there exists an input for which the path is traversed during program execution, otherwise the path is unfeasible.

Mutation testing (strong):

Mutation testing creates a number of mutant programs, which differ from the original in one small way (i.e. each possessing a fault). The original test data are run through the mutant programs. If the test data detect the difference in the mutant program (i.e. they reveal the fault by producing different output from the original program) then the mutant is said to be dead. If, on the other hand, one of the mutants does not produce different results, then the test data need to be examined and augmented to reveal the fault and kill the live mutant. The advantage of strong mutation testing is it shows that absence of particular faults, while the weakness is that it is computational expensive to carry out.

Mutation testing (weak):

Weak mutation testing takes some component of a program and creates a number of mutants of this component. Test data are then run through the mutants and if, on at least one execution, the results from the mutants and the original are different, then the test data are considered to be adequate. Five components are identified in weak mutation testing: data access (or variable reference), data storage (or variable assignment), arithmetic expression, arithmetic relation and Boolean expression. Weak mutation is computational cheaper than strong mutation in that it is usually not necessary to physically generate all the mutants, while the weakness is it is not reliable for the program as a whole, and the adequacy of the data is local to the component under test.

There are some other testing criteria in software testing domain, i.e., DD path (decision-to-decision path), condition testing, multiple condition testing, level-i paths, basic path testing, linear code sequence and jump (LCSAJ), data flow testing, domain testing, partition analysis, equivalence partitioning, boundary value analysis, cause-effect graphing and category-partition testing. Complete and detailed introductions white-box testing criteria can be found in Roper [1994].

1.2. Test Data Generation

Extensive testing can only be carried out by an automation of the test process. Since test data generation is the major and most crucial part of test process, the benefits of applying automatic test data generation are a reduction in time, effort, labor and cost for software testing.

1.2.1. Static Test Data Generation

Static test data generation method analyzes the software under test without executing the code, either manually or automatically. Symbolic execution and evaluation, introduced by Howden [1975], Darringer [1978] and Clarke [1985], is a typical static method for generating test data. Symbolic execution provides a functional representation of the path in a program and assigns symbolic names for the input values and evaluates a path by interpreting the statements and predicates on the path in terms of these symbolic names. All static approaches are hampered by the limitations of symbolic execution - loops are only analyzed in terms of their entry and exit points or have to be unrolled, analysis of arrays is memory intensive or must be approximate and the problems of pointers and dynamic memory are hard to address.

1.2.2. Dynamic Test Data Generation

Dynamic techniques overcome the problems experienced by static techniques by actually executing the software and exploiting the information that is only available at run-time. There are three types of dynamic techniques:

1.2.2.1. Data Specification Generators

Deriving test data from specification belongs to the '*black-box*' testing method. Such a strategy generates test cases and test data e.g. from formal Z specification, Yang [1995]. The test data can then be applied to software and the effectiveness can be measured, e.g. using ADATEST in Gallagher [1997] (ADATEST is an automatic testing system for Ada software which measures for example the percentage of statements executed or branches covered). A disadvantage is that a formal specification for the software may not often exist.

1.2.2.2. Random Test Data Generation

Random test data generators, Sturgis [1985] and Voas [1991], select arbitrarily test data from the input domain and then these test data are applied to the program under test. The automatic production of random test data, drawn from a uniform distribution, should be the default method against which other systems should be judged, Ince [1987]. It is suggested that the distribution of selected input data should have the same probability distribution of inputs that will occur in actual use (operational profile or distribution which occurs during the real use of the software) in order to estimate the operational reliability.

Random testing only requires a random number generator and a small amount of software support. The adequacy of random data is very dependent on the interval (range) from which the data is generated. Data from poorly chosen intervals are much worse than those from well-chosen intervals. A disadvantage of random testing is to satisfy equality values, which are difficult to generate randomly. Random testing is pretty good for the final testing stage of software. Duran [1981] recommended a mixed final testing, starting with random testing, followed by a special value testing method (to handle exceptional cases). Ince [1987] reported that random testing is a relatively cheap method of generating initial test data.

1.2.2.3. Pathwise Test Data Generation

Pathwise test data generators, Korel [1990] and DeMillo [1991], are systems that test software using a testing criterion, which can be path coverage, statement coverage, branch coverage, etc. The system automatically generates test data to the chosen requirements. A pathwise test generator consists of a program control flow graph construction, path selection and test data generation tool.

Dynamic approaches of test data generation have been shown to perform effectively for a number of structural testing-criteria and overcome many of the weaknesses of static techniques. Their ability to work well in the presence of arrays allows them to generate test data for a wider class of software. However, most of the approaches are far from perfect or empirical. Using simple search techniques or single search technique restrict the application of the dynamic approaches on test data generation. Most of current approaches ignore the fact that none single test data search technique is perfect – one technique works very well to some tested programs may be very weak in testing other programs.

2. Genetic Algorithm (GA)

2.1. Outline of GA

The study of genetic algorithms is originated with John Holland [1975] in the mid-1970s, and it is investigated by Back [2000], Fogel [1995] and Goldberg [1989]. Further studies of using genetic algorithms on test data generation were made by Korel [1990], Ferguson [1997], Gallagher [1997], Pargas [1999], Tracey [2000], Michael [2001] and Godefroid [2002].

The name “genetic algorithm" comes from the attempt to model the natural genetic evolutionary process. The basic idea of genetic algorithms is to represent candidate solutions to the concrete problem using a vector of components known as chromosomes, and then model selective breeding to obtain “offspring" that have characteristics inherited from each parent, just like natural evolution.

Genetic algorithms work by maintaining a population of chromosomes, and each chromosome in the population has an associated fitness to determine which chromosomes are used to form new ones in the competition process (called selection). Successive populations (known as generations) are evolved with certain probability using genetic operations - parents are combined (crossover) and new genetic information is introduced (mutation). Genetic algorithms have had a great measure of success in search and optimization problems. The reason for a great part of their success is their ability to exploit the information accumulated about an initially unknown search space in order to bias subsequent searches into useful subspaces, i.e. adaptation, which is the key feature, particularly in large, complex and poorly understood search spaces, where classical search techniques are inappropriate.

Only general introduction and major techniques of genetic algorithms are included in this chapter, and complete introductions are provided in Goldberg [1989] and Mitchell [1996].

An outline of the genetic algorithm search process is shown as follows:

```
Procedure Genetic Algorithm is
Current Population, Parents, Offspring:
Array (1. .N) of Chromosomes;
Begin
    INITIALIZE (Current Population);
        - Assign initial values to each member of  the population
    CALC FITNESS (Current Population);
        - Calculate the fitness for each member of the
          population, related to the objective
        - Function for the solution represented by the
          chromosomes
Loop
    SELECT SURVIVORS (Current Population, Offspring);
        - Select which members of the population will survive to
          be offspring directly
    SELECT PROSPECTIVE PARENTS (Current Population, Parents);
        - Select which members of the population are to be used
          as parents
    RECOMBINE (Offspring, Parents);
        - Combine parents to produce offspring
    MUTATE (Offspring);
        - Stochastically mutate the new offspring to introduce
          diversity
    CALC FITNESS (Offspring);
Exit when STOP CRITERION;
End loop;
End Genetic Algorithm;
```

The generic decisions fall into three classes – representation issue, selection mechanism and genetic operations.

2.2. Representation Issue

Representation is a key issue in GA work because GAs directly manipulate a coded representation of the problem and because the representation can severely limit efficiency of GAs. There are mainly two representations using in GAs: Binary coding and real coding.

***Binary coding*:** To represent the elements of a search space $S=S_1$ x...x S_n by means of the binary alphabet, a function cod_i: $S_i \rightarrow \{0,1\}^{Li}$, $L_i \in N$, should be specified, which codes each element in S_i using binary strings of length Li. An element $x=(x_1, \ldots, x_n) \in S$ $(x_i \in S_i)$ is represented by linking together the coding of each one of its components cod (x)=cod1 $(x_1) \ldots cod_n (x_n)$, Herrera [1996] .

***Real coding*:** it would be particularly natural to represent genes directly as real numbers for optimization problems of parameters with variables in continuous domains. Then a chromosome is a vector of floating point numbers, the precision of which will be restricted only to that of the computer with which the algorithm is carried out. The size of the chromosomes is kept the same as the length of the vector, which is the solution to the problem.

Fixed-length and binary coded strings for the representation solution have dominated GA research since there are theoretical results that show that they are the most effective ones, Goldberg [1991], and they are easy to implement. However, GA's good properties do not stem from the use of bit strings. In many cases, GA practitioners devised non-binary representations, and proved them to be more natural for specific application problems.

The advantages of real coding are as follows:

1. Real parameters makes it possible to use large domains for the variables, which is difficult to achieve in binary implementations where increasing the domain would mean sacrificing precision, assuming a fixed length for chromosomes.

2. Real parameters' capacity to exploit the graduality of the functions with continuous variables.

3. It is very close to natural formulation of many problems.

4. Real coding allows the domain knowledge to be easily integrated in the real coded GA for the case of problems, and more complex operations are possible.

2.3. Selection Mechanism

Selection methods decide which solutions will become parents and also which solutions will survive into later generations. Parent selection in Holland's [1975] original work was based on fitness - solutions with a higher fitness value were more likely to be selected as parents, known as roulette wheel sampling. This aims to mirror the parent selection process in nature, where fit individuals have higher probabilities to survive and become parents. The idea is that by combining the information of fit parents even fitter offspring can be produced. However, the selection of parents is subject to sampling errors and this can lead to a significant difference between the actual and expected number of times a solution is used as a parent. De Jong [1975] devised an expected value model. This determines the expected number of times each solution will be selected as a parent and ensures the actual selection is closely in line with this expectation. Barker [1985] pointed out that these fitness-based techniques for selecting parents could result in a strong bias towards a few very fit solutions. This can cause the search to converge too quickly and get trapped in a local minimum. Barker suggests ranking selection to overcome this problem. Here, solutions are ranked according to fitness. The parent selection is then based on rank, with a bias towards the higher ranks. This abstracts away from the actual values of the fitness, but still allows fitter solutions to be selected more frequently. Another selection scheme discussed by Goldberg and Deb [1991] is tournament selection, which selects k individuals from the current population with uniform possibility, and has them do the "tournament". Often tournaments are held only between two individuals. There are also many variations suggested on these techniques.

2.4. Genetic Operations

After selection has been carried out, the construction of the intermediate population is complete and the operators of crossover and mutation are applied.

The crossover operator is a method for sharing information between chromosomes; it combines the features of two parent chromosomes to form two offspring, with the possibility that good chromosomes may generate better ones. The crossover operator is not usually applied to all pairs of chromosomes in the intermediate population. A random choice is made, where the likelihood of crossover being applied depends on probability defined by a crossover rate, the crossover probability. The crossover operator plays a central role in GAs, in fact, it may be considered to be the one of the algorithm's defining characteristics. Definitions for this operator depend on the particular representation chosen. The mutation rate operator arbitrarily alters one or more components, or genes of a selected chromosome so as to increase the structural variability of the population, thus introduce some sort of randomness into the chromosome. The role of mutation in GAs is to restore lost or unexplored genetic material into the population to prevent the premature convergence of GAs to suboptimal solutions; it insures that the probability of reaching any point in the search space is never zero. Each gene of every chromosome in the population undergoes a random change according to a probability defined as a mutation rate, the mutation probability.

The implementation of the genetic operators is partially dependent on the solution encoding method. Although much of the research into genetic operators has focused on manipulation of binary strings, there are some researchers working on real-coded operators. Hereafter are some operators using for binary-coded operation and real-coded operation:

2.4.1. Binary Coding Operators

- Crossover operators:

Simple crossover (Holland [1975]): given two chromosomes C_1=($c_1^1 \cdots c_L^1$) and C_2=($c_1^2 \cdots c_L^2$), the offspring H_1=($c_1^1,\ldots,c_i^1,c_{i+1}^2,\ldots,c_L^2$) and H_2=($c_1^2,\ldots,c_i^2,c_{i+1}^1,\ldots,c_L^1$) are generated, where *i* is a random number belonging to {1, ... , L-1}. Figure 2.1 shows an example of this operator's application.

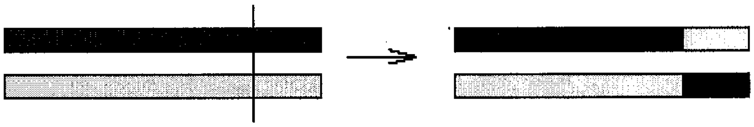

Figure 2.1 An Example of Simple Crossover

N-point crossover (Eshelman [1989]): n crossover points are randomly selected and the corresponding segments of the parents are exchanged for generating the offspring.

Uniform crossover (Syswerda [1989]): the values of each gene in the offspring are determined by the uniform random choice of the values of this gene in the parents.

- Mutation operation (Holland [1975]: given a chromosome, a gene is randomly chosen and its value is swapped; "1" or "0" and vice versa. (See figure 2.2)

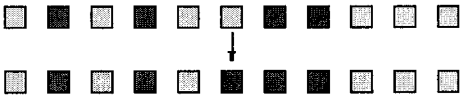

Figure 2.2 An Example of Mutation Operation

2.4.2. Real Coding Operators

- Crossover:

Assume that C1=($\overset{1}{c_1}\cdots\overset{1}{c_n}$) and C2=($\overset{2}{c_1}\cdots\overset{2}{c_n}$) are two chromosomes that have been selected to apply the crossover or mutation operator to them.

Flat crossover (Radcliffe [1991]):

An offspring H=(h_1,...,h_i,...,h_n), is generated, where h_i is a randomly (uniformly)chosen value of the interval [$\overset{1}{c_i}$,$\overset{2}{c_i}$].

Simple crossover (Wright [1991]):

A position $i \in \{1,2,\ldots,n-1\}$ is randomly chosen and the two new chromosomes are built.

$H_1=(c_1^1, c_2^1 \ldots, c_i^1, c_{i+1}^2, \ldots, c_n^2)$

$H_2=(c_1^2, c_2^2 \ldots, c_i^2, c_{i+1}^1, \ldots, c_n^1)$

Arithmetical crossover (Michalewicz, [1992])

Two offspring, $H_k=(h_1^k,\ldots,h_i^k,\ldots,h_n^k)$ k=1, 2, are generated, where $h_i^1 = \lambda c_i^1 + (1-\lambda) c_i^2$ and $h_i^2 = \lambda c_i^2 + (1-\lambda) c_i^1$. λ is a constant (uniform arithmetical crossover) or varies with regard to the number of generations made (non-uniform arithmetic crossover).

BLX-α crossover (Eshelman [1993]):

An offspring is generated: $H=(h_1,\ldots,h_i,\ldots,h_n)$, where h_i is a randomly (uniformly) chosen number of the interval $[c_{min}-I*\alpha, c_{max}+ I*\alpha]$, $c_{max}=\max(c_i^1, c_i^2)$, $c_{min}=\min(c_i^1, c_i^2)$, $I= c_{max}- c_{min}$. The BLX-0.0 (α=0.0) crossover is equal to the flat crossover.

Linear crossover (Wright [1991]):

Three offspring $H_k=(h_1^k,\ldots,h_i^k,\ldots,h_n^k)$ k=1,2,3, are built, where $h_i^1 = \frac{1}{2}c_i^1 + \frac{1}{2}c_i^2$, $h_i^2 = \frac{3}{2}c_i^1 - \frac{1}{2}c_i^2$ and $h_i^3 = \frac{1}{2}c_i^1 + \frac{3}{2}c_i^2$. With this type of crossover, an offspring selection mechanism is applied, which chooses the two most promising offspring of the three to substitute their parents in the population.

Discrete crossover (Mühlenbein [1993])

h_i is a randomly (uniformly) chosen value from the set $\{c_i^1, c_i^2\}$.

Extended line crossover (Mühlenbein [1993])

h_i=$c_i^1+\alpha(c_i^2-c_i^1)$ and α is a randomly (uniformly) chosen value in the interval [-0.25,1.25].

Extended intermediate crossover (Mühlenbein [1993])

h_i=$c_i^1+\alpha_i(c_i^2-c_i^1)$ and α_i is a randomly (uniformly) chosen value in the interval [-0.25,1.25]. This operator is equal to the BLX-0.25.

Wright's heuristic crossover (Wright [1990])

Let's suppose that C_1 is the parent with the better fitness. h_i=$r\cdot(c_i^1-c_i^2)+c_i^1$ and r is a random number belonging to [0,1].

Linear BGA crossover (Schlierkamp-Voosen [1994])

Under the same consideration as above, hi=$c_i^1 \pm rang_i \cdot \gamma \cdot \Lambda$, where Λ=$\dfrac{c_i^2-c_i^1}{\left\|c_1-c_2\right\|}$. The "-" sign is chosen with a probability of 0.9. Usually, rang$_i$ is 0.5 times the domain of the selected variable, and γ= $\sum_{k=0}^{15}\alpha_k 2^{-k}$ where $\alpha_i \in \{0,1\}$ is randomly generated with p(α_i=1)= $\dfrac{1}{16}$. This operator is based on Mühlenbein mutation, Mühlenbein [1993].

Fuzzy Connection Based Crossover (FCB) (Herrera [1994])

In short, the interval of action of the gene i[a_i, b_i] may be divided into three regions [a_i, α_i], [α_i, β_i], [β_i, b_i], where good descendents may be obtained; even considering a region [α_i', βi'] with α_i'<=α_i and βi'>=βi would seem reasonable. Figure 2.3 shows this graphically.

Detailed introduction is provided in Herrera [1994].

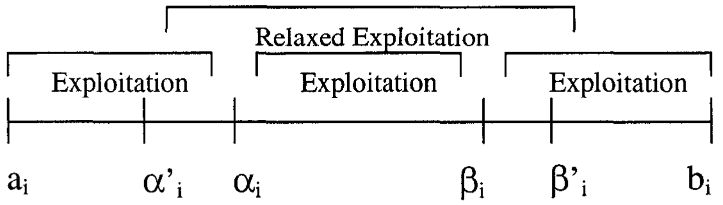

Figure 2.3 Fuzzy Connection Based Crossover

For each type of crossover operations presented earlier in this section, Figure 2.4 shows the effect on gene i.

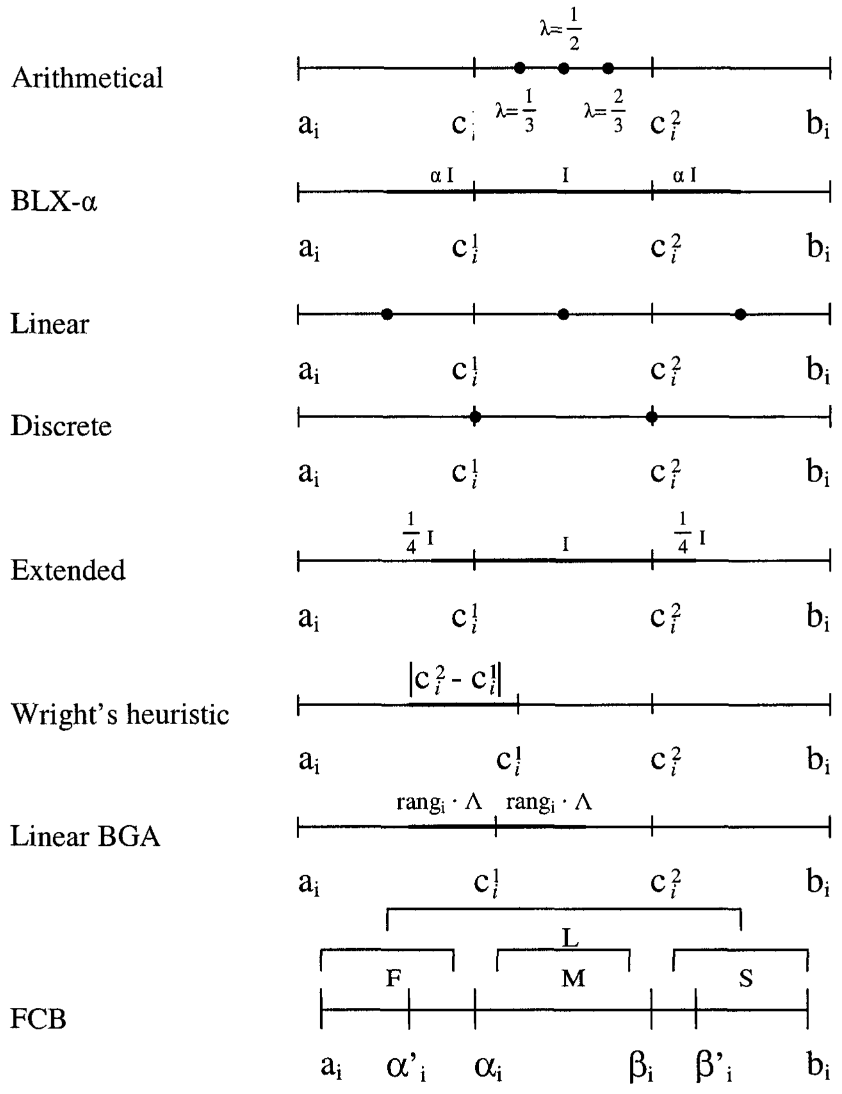

Figure 2.4 Gene i for Different Types of Crossover Operators

- Mutation:

Suppose that $C_1=(c_1,\ldots,c_i,\ldots,c_n)$ is a chromosome and $c_i \in [a_i, b_i]$ is a gene to be mutated. c_i' is the result that generated from a mutation operation.

Random mutation (Michalewicz [1992])

c_i' is a random (uniform) number from the domain $[a_i, b_i]$.

Non-uniform mutation (Michalewicz [1992])

If this operator is applied in a generation t, and g_{max} is the maximum number of generations then

$$c_i' = \{ \begin{matrix} c_i + \Delta(t, b_i - c_i) & if \tau = 0 \\ c_i - \Delta(t, c_i - a_i) & if \tau = 1 \end{matrix}$$

with τ being a random number which may have a value of zero or one, and

$$\Delta(t,y) = y(1 - r^{(1-\frac{t}{g_{mzx}})^b}),$$

where r is a random number from the interval [0,1] and b is a parameter chosen by the user, which determines the degree of dependency on the number of iterations. This function gives a value in the range [0,y] such that the probability of returning a number close to zero increases as the algorithm advances. The size of the gene generation interval shall be lower with the passing of generations. This property causes this operator to make a uniform search in the initial space when t is small and very locally at a later stage, favoring local tuning.

Real Number Creep (Davis [1991])

When a continuous function is optimized with local maximums and minimums and, at a given moment in time, a chromosome is obtained which is situated in a good local maximum, it would be interesting to generate other chromosomes around this chromosome in order to come close to the peak point of such a maximum. In order to do this, we may slide the chromosome into a value that increases or decreases it by a small random quantity. The maximum slide allowed is determined by a parameter defined by the user. Different instances of this operator have been presented, such as the Guaranteed-Big-Creep and the Guaranteed-Little creep (Davis [1989]) and the small creep and the large creep (Kelly [1991]). The difference between these operators lies in the value of the maximum slide allowed.

Mühlenbein's mutation (Mühlenbein [1993])

$$c_i' = c_i + rang_i \cdot \gamma,$$

where *rang*$_i$ defines the mutation range and it is normally set to $0.1*(b_i-a_i)$. The + and – sign is chosen with a probability of 0.5 and

$$\gamma = \sum_{k=0}^{15} \alpha_k 2^{-k},$$

$\alpha_i \in \{0, 1\}$ is randomly generated with $p(a_i=1)=1/16$.

Values in the interval $[c_i\text{-}rang_i, c_i\text{+}rang_i]$ are generated using this operator, with the probability of generating a neighborhood of c_i being very high. The minimum possible proximity is produced with a precision of $rang_i \cdot 2^{-15}$.

There are other two operators, which are based on Mühlenbein mutation, and the only difference is how to calculate the γ:

Discrete modal mutation (Voigt [1994])

$$\gamma = \sum_{k=0}^{\pi} \alpha_k B_m^k,$$

with $\pi = \left\lfloor \frac{\log(rang_{\min})}{\log(B_{m)}} \right\rfloor$. B_m >1 is a parameter called the base of the mutation and $rang_{min}$ is the lower limit of the relative mutation range.

Continuous modal mutation (Voigt [1994])

$$\gamma = \sum_{k=0}^{\pi} \alpha_k \phi(B_m),$$

with ϕ (z_k) being a triangular probability distribution with $\frac{B_m^k - B_m^{k-1}}{2} \leq z_k \leq \frac{B_m^{k+1} - B_m^k}{2}$.

We should point out that once parents have been selected and offspring formed after the crossover and mutation, an additional selection strategy should be used to determine the survivors for the next generation. In Holland's original work, all of the offspring survived to become the next generation. The elitist model introduced by De Jong [1975] attempts to prevent good solutions from one generation being lost in later generations, due to crossover or mutation. This is achieved by combining the best of the current generation and the offspring.

The parameters for a genetic algorithm include the population size, number of parents, number of offspring, rate of crossover and rate of mutation. Most applications of genetic algorithms rely on setting these parameters by trial-and-error.

3. Program Dependence Graphs

3.1. Program Dependence Graphs (PDG)

Program dependence graphs were introduced by Kuck [1981] as an intermediate program representation well suited for performing optimizations (Ferrante [1987]), vectorization, and parallelization. PDG represents a program as a graph in which the nodes are statements and predicate expressions, and edges connected to a node represent both control conditions on which the execution of the operations depends and the data values on which the node's operations depend.

Dependences in a program arise as a result of two separate effects. First, dependence exists between a statement and the predicate whose result immediately controls the execution of the statement. For example,

```
if (A)                              N1
          B=C+D;                    N2
else B=C * D                        N3
```

N2 and N3 depend on predicate A since the value of A determines whether N2 or N3 is executed. Dependences of this type are *control dependences*. Second, dependence exists between two statements whenever a variable appearing in one statement may have an incorrect value if the two statements are reversed. For example,

```
A=B+C;                              N1
D=A * E;                            N2
```

N2 depends on N1, because executing N2 before N1 would result in N2 using incorrect value of A. Dependences of this type are *data dependences or flow dependences*.

Constructing a program's control dependence subgraph and data dependence subgraph, which are defined in terms of control flow graph, are the two steps of constructing a PDG.

3.1.1. Control Dependence

In this section, we define control dependence in terms of a control flow graph (CFG), which has a unique entry node ENTRY and a unique exit node EXIT.

Definition 1. A node V is *post-dominated* by a node W in a CFG if every directed path from V to exit (not including V) contains W. The definition of post-dominance does not include the initial node on the path, so a node never post-dominates itself, but a node (if it is a predicate node) can post-dominate its L-branch (where L-branch could be either true or false branch).

Definition 2. X and Y are nodes in a CFG. Y is *control dependent* on X iff

1) Y post-dominates L-branch of X and

2) X is not post-dominated by Y.

If Y is control dependent on X, then X must be a predicate. Following one branch from X always results in Y being executed, while taking another branch may result in Y not being executed. Condition 2 is always satisfied when X and Y are the same node, which allows loops to be correctly accommodated by the definition.

Figure 3.1 depicts program **SUM** on the left, its control-flow graph in the center, and its control-dependence graph on the right. T represents TRUE and F for FALSE.

In the example, "printf (sum)" and "printf (i)" post-dominate the node "i<11", which means no matter i<11 is true or false, they are going to be executed. We can see from the CDG that all nodes are control dependent on node "Enter" except for nodes "sum=sum+1" and "i=i+1", which are control dependent on node "i<11". Besides, node "i<11" is control dependent on its true branch.

```
int main () {
  int sum = 0;
  int i=1;
  while (i<11) {
    sum=sum+i;
    i=i+1;
  }
  printf ("%d\n",sum);
  printf ("%d\n",i);
}
```

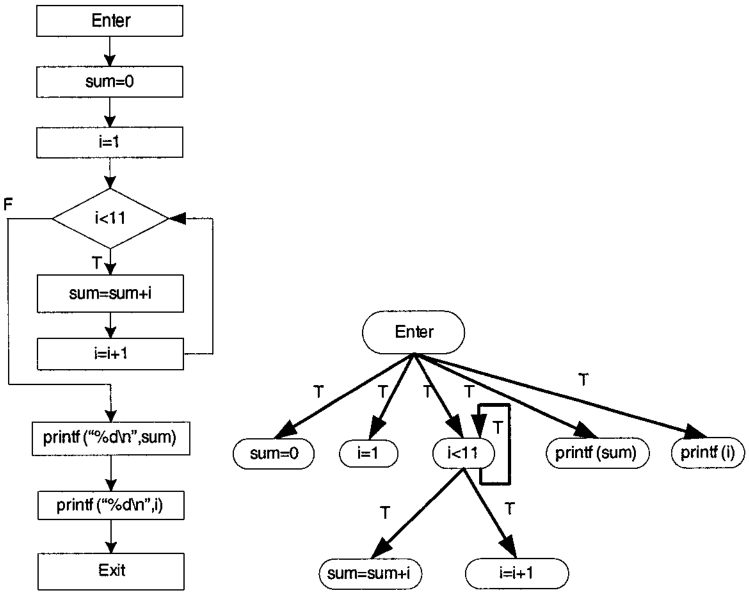

Figure 3.1 Program SUM on the left, its control-flow graph (CFG), and its control dependence graph (CDG) on the right.

3.1.2. Data Dependence

Definition 3. X and Y are nodes in a CFG. Y is data dependence on X iff

1) X assigns to variable Z, and Y uses Z, and

2) there is a path in the CFG from X to Y that does not include an assignment to Z (excluding X and Y).

Figure 3.2 shows the program dependence graph of the SUM program, with the data dependence graph and control dependence graph together. The black straight lines with arrows represent control dependence relations and the grey curved lines with arrows are for data dependence relations.

In the PDG, variable "sum" in the node "sum=sum+i" is data dependent on the initial assignment of sum and i, and it is also data dependent on the values of sum and i in the last iteration of the while loop.

```
int main () {
  int sum = 0;
  int i=1;
  while (i<11) {
     sum=sum+i;
     i=i+1;
  }
  printf ("%d\n",sum);
  printf ("%d\n",i);
}
```

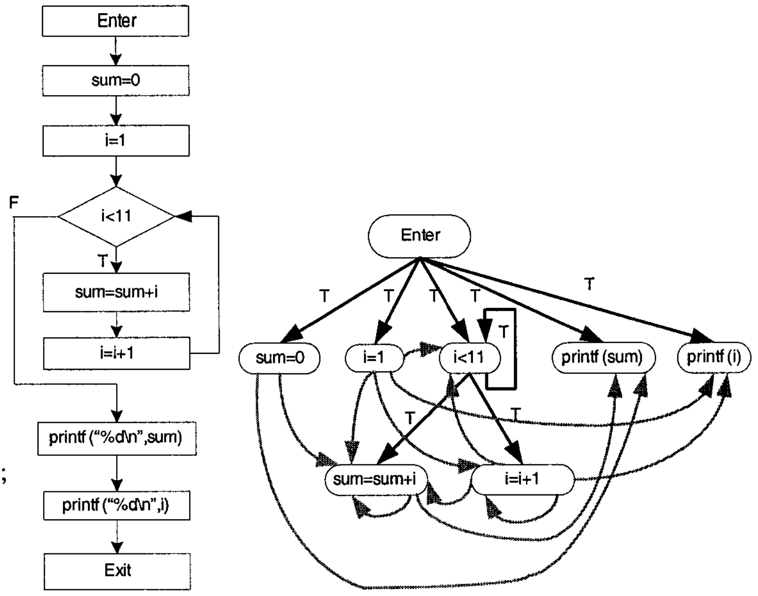

Figure 3.2 Program SUM (left), CFG (middle) and Control Dependence Graph and Data Dependence Graph (right)

3.2. A Tool Using PDG

Constructing PDG manually is a time-consuming job, and practically impossible when programs keep getting bigger and complicated. There are tools that automatically generate PDGs, provide PDG information that users need, and provide flexible and powerful queries on PDGs.

A commercial tool called CodeSurfer, introduced in Anderson [2001] and GrammaTech [1999], is used in the thesis project to assist us to get PDG information that we need of the programs under test and do queries on PDGs. CodeSurfer is a powerful source code analysis and navigation tool, providing easy and precise navigation and understanding of source code. CodeSurfer has many uses including program understanding, maintenance, impact analysis, debugging, reengineering, and reuse.

CodeSurfer works like a compiler - it parses and analyzes all the source-code components of the system. The dependences between statements in the program are computed first by analyzing direct dependences, then by analyzing indirect dependences. The results are stored in a data structure called a *System*

Dependence Graph (SDG), which is a comprehensive system-level graph including multiple PDGs for different procedures with intraprocedural control and data dependence information, Horwitz [1990]. The elements of the SDG are the declarations, statements and conditions. They are linked together to describe their data-dependence and control-dependence relationships. Sophisticated algorithms are used to traverse the SDG to answer queries. The result of a query evaluation is a set of SDG elements that can then be mapped back to locations in the source code.

CodeSurfer not only provides graphic user interface but also can be programmed, extended, customized, and integrated with other applications using its scripting language. Its scripting language is based on Scheme, a general purpose programming language, with some extended additional data types for representing the dependence-graph representation of C programs.

With CodeSurfer, users can easily get PDG information and do queries on PDGs. By running CodeSurfer scripts, results can be got without using the GUI of CodeSurfer, and with the API of CodeSurfer, users can integrate CodeSurfer with other testing applications to construct a testing tool, which is capable of using program PDG information to test software.

4. Previous Research on Test Data Generation

4.1. TESTGEN by Korel

The traditional dynamic test data generation paradigm, implemented by Korel [1990] and Korel [1996] with a tool called TESTGEN, uses the idea that test requirements can be transformed and treated as functions. If some test requirement is not satisfied, the test requirement can be transformed to a function, and data collected during the execution of the program can be used to determine the values of the function, which represent how close different test cases are to the function minimum. Therefore, the problem of satisfying test requirements by generating test data is converted to the well-understood problem of function minimization.

For example, if the requirement is to traverse the true branch of *if (pos>=21)* on line 324 in a program, suppose $pos_{324}(x)$ represents the value of *pos* recorded on line 324 when the program is executed with the input x, the function $\Im(x) = \begin{cases} 21 - pos_{324}(x), & if\, pos_{324}(x) < 21; \\ 0, & otherwise \end{cases}$ is minimal when the true branch is taken on line 324. Thus, the problem of test data generation is reduced to the problem of the function minimization, which is that in order to find the proper input to traverse desired branch, we must find a value of x that minimizes $\Im(x)$.

In TESTGEN system of Korel [1990], the tested program is executed on a seed input, and subsequent action depends on whether the execution reaches the sections of code where the test requirement is supposed to hold. If it does, function minimization methods can be applied to find a needed input value. If the code is not reached, a subgoal is created to bring about the conditions necessary for function minimization to work. The subgoal consists of redirecting the flow of control so that the desired section of code will be reached. TESTGEN finds branches that are responsible for directing the flow of control away from the desired location, and attempts to modify the seed input to force the control of execution to the desired direction. The approach to the new subgoal can be treated as that of its parent goal. Likewise, more subgoals may be created to satisfy the first subgoal. This recursive creation of subgoals is called chaining in Korel [1990] and Ferguson [1996].

Korel's approach has the advantage on freely choosing which path it wants, and in the TESTGEN system, heuristics are used to select the path that seems most likely to have an impact on the target condition.

4.2. ADTEST system by Gallagher

Besides Korel's TESTGEN, ADTEST system of Gallagher [1997] specifies an entire path in advance, and then the goal is to find an input that executes the desired path. Since it is known which branch must be taken for each condition on the path, all of these conditions can be combined in a single function whose minimization leads to an adequate test input.

For example, if the desired path requires taking the true branch of the condition *if (b>=10) ...* on line 11 and taking the false branch of the condition *if (c>=8)* on line 18, then one can find an adequate test input by minimizing the function $\Im_1(x) + \Im_2(x)$, where

$$\Im_1(x) = \begin{cases} 10 - b_{11}, & if b_{11} < 10; \\ 0, & otherwise \end{cases}$$

$$\Im_2(x) = \begin{cases} c_{13} - 8, & if c_{13} > 8; \\ 0, & otherwise \end{cases}$$

Unfortunately, this function cannot be evaluated until line 10 and line 18 are both reached. Therefore, the ADTEST system begins by trying to satisfy the first condition on the path, adding the second condition only after the first condition has been satisfied. As more conditions are reached, they are incorporated in the function that the algorithm seeks to minimize.

4.3. GADGET by Michael

Another software test data generator, GADGET introduced by Michael [2001], applies genetic algorithm on the test data generation problem, which is not used in most previous test data generation systems. GADGET uses condition-decision coverage that leads to two test requirements for each condition in the code, namely, each condition must be true and false for at least once respectively. Condition-decision coverage also requires that each branch of each decision be taken at least once.

Before starting the GA, a seed input is used to execute the program, and after the first execution, a coverage table is initialized with the coverage status of each condition or decision for the purpose of tracking if a condition or a decision is tested or not. (See the Table 4.1 below). After this, the algorithm uses the coverage table to select a series of test requirements in turn.

Table 4.1 An Example of the Coverage Table

	Status	
Condition/Decision	TRUE	FALSE
1	X	X
2	-	-
3	-	X
4	X	-
5	X	-
6	-	-

For each test requirement, the GA is initialized and attempts to satisfy the given requirement. Whenever the GA generates an input that satisfies a new test requirement, no matter whether it is the one that GA is currently working on, the new test input is recorded for the future use and the coverage table is updated. The test data generator continues to iterate over the test requirements until no further progress can be made. GADGET uses a commercial coverage analysis tool (DeepCover) to measure the condition-decision coverage. Michael's approach is different from TESTGEN and ADTEST in the way that it doesn't concentrate on one specific path to the desired location. Instead, it delays the conditions that haven't been reached, and works on the condition that has been reached and where either the true or false branch hasn't been satisfied yet. When the GA works on a certain requirement, many other requirements are often coincidentally satisfied. This phenomenon is named "serendipitous satisfaction" by Michael [2001], and it plays an important role in GADGET.

For each of not completely satisfied test requirement, a fitness function $\Im$ is generated to map the problem to a minimization problem. Table 4.2 shows how $\Im$ is calculated for some typical relational operators in GADGET.

Table 4.2 Operators for Fitness Functions

Decision type	Example	Fitness function
Inequality	if (c>=d)...	$\Im(x)=\begin{cases} d-c, & if \quad d \geq c; \\ 0, & otherwise \end{cases}$
Equality	if (c=d)...	$\Im(x)=\lvert d-c \rvert$
true/false value	if (c)...	$\Im(x)=\begin{cases} 1000, & if \quad c = FALSE; \\ 0, & otherwise \end{cases}$

There are some experiments done on 9 simple programs with GADGET. They are:

- Binary search
- Bubble sort
- Number of days between two dates
- Euclidean greatest common denominator
- Insertion sort
- Computing the median

- Quadratic formula
- Warshall's algorithm
- Triangle classification

The condition-decision coverage results are in Table 4.3:

Table 4.3 Condition-decision Coverage of GADGET

Program	Random	GA
Binary search	80	70
Bubble sort 1	100	100
Bubble sort 2	100	100
Number of days between two dates	87.5	100
Euclidean greatest common denominator	100	100
Insertion sort	100	92.3
Computing the median	100	100
Quadratic formula	75	75
Warshall's algorithm	91.7	100
Triangle classification	48.6	94.29

The advantage of GADGET is that it uses GA instead of gradient descent used by TESTGEN and ADTEST to avoid to be stuck at the local minima. Moreover, GADGET greatly simplifies the dynamic test data generation by skipping complicated control flow analysis for a specific path, with the hope that most of the conditions in the coverage table would be partially satisfied by the initial input or coincidentally partially satisfied when GA is working on some other conditions.

However, there are some problems with Michael's algorithm. The test data generator GADGET depends heavily on the serendipity of whether a location of the tested program is reached. In GADGET, only those conditions that have been reached or partially satisfied in the condition-decision coverage table are converted to functions. Other conditions that haven't been reached are left behind until they are reached by coincidence satisfaction of some input that generated by the GA trying to satisfy other conditions. Since the serendipity satisfaction plays an important role in GADGET, the tool may ignores those unreached conditions throughout the algorithm and may give them up without any trials at the termination of the optimization procedure. This approach greatly simplifies the algorithm, while still enabling it to succeed with some small programs. Unfortunately, there are many commercial programs or procedures with complex control flow paths and cascaded data dependence between variables, and it is hard or often impossible to find solutions only by coincidences. In simple programs, this approach may reach 100% coverage, but when programs get larger, the chance that each condition in the coverage table gets reached is very low. Hence, GADGET has to abandon some hard-to-reach conditions or branches, reducing the overall coverage. In Michael's approach, it trades off the coverage rate with the simplicity of the algorithm. Surprisingly, Michael's approach has worse coverage on 2 programs than random testing, and there are 5 programs with the same coverage percentages of GADGET and random testing. There are some reports from other researchers about the poor coverage results of using Michael's algorithm. When we carefully analyze GADGET, it happens in most cases that when programs get complex, what GADGET really does is generating random values in repeated attempts to satisfy the functions.

Since performance of GADGET mostly depends on the performance of GA, the tool inevitably inherits the weakness of the GA. The technique used by GADGET to define the fitness function makes the tool inadequate to deal with conditions with Boolean variables or enumerated types. For example, if GADGET is trying to exercise the true branch of the condition

if (*windy*)...

It simply makes $\Im(x)$ equal to the absolute value of *windy*. This makes $\Im(x)$ zero when the condition is true and positive otherwise. However, if windy only takes on two values of 0 and 1, then the fitness function can only have two values as well. Two-valued fitness function does not allow the genetic algorithm to distinguish between different inputs that fail to satisfy the test requirement, thus genetic search cannot apply a preference on one set of inputs over other sets as they have the same positive value as their fitness. With serendipity satisfaction that GADGET uses, the chance of the GA to traverse the true branch of conditions containing Boolean variables is no more than the chance of random search. Some of the failures reported in Michael [2001] are:

```
for (index=begin; index<=end && !termination; index++)
if (((T<=0.0)||(0.0==Gain)))
if(o_5)
if (o_5)
if (((o_5>0.0) && (o<0.0)))
if (FLARE)
if (FLARE)
if (DECRB)
```

Enumerated types are a little better than binary variables for GAs to handle, but GAs still have difficulty in guiding the search towards the minimization, because only limited values are available for GAs to evaluate the fitness of different inputs.

Although path selection is not vital to GADGET, it may still be the case that some execution paths are better than others for satisfying a particular test requirement. If a path selection algorithm could be introduced to GADGET, it wouldn't be difficult for the genetic search algorithm to favor solutions using the selected paths. Another drawback of GADGET is that because all the inputs that cannot reach the condition that the tool is trying to satisfy get the same low values as their fitness values, there is no way for GA to bias the inputs that are closer to reaching the condition, and hence the possibility of breeding more inputs to actually reach the condition is reduced.

4.4. Constraint-based Testing (CBT) by Offutt

Another test data generator, Godzilla by Offutt [1988], presents a technique for using mathematical constraints for testing, which is called Constraint-Based Testing (CBT), DeMillo [1991]. CBT adopts weak mutation testing criterion: introducing faults to the program under test by creating mutants of the program, and generating test data to kill the mutants by revealing the faults and distinguish the mutant from the original. Godzilla integrates Mothra Software Testing Environment Project to generate the mutants. Mothra uses twenty-two mutation operators for testing FORTRAN programs and over seventy for testing C programs. These operators are designed to simulate the common mistakes that human programmers might take.

Although with the mutation operators it is clear that the mutated statement of the mutant program is different from the counterpart of the original program, it doesn't guarantee the state difference after the execution of the mutated statement, and the program and its mutant will never differ if the states of the two programs don't differ after the execution of the mutated part, because the program and its mutant are syntactically equal except for the mutated statement. In order to cause the wished state different to kill the mutant, a term "necessity constraint" is defined to make sure of the incorrect output of the mutated statement. Table 4.4 shows an example of constructing necessity constraint for the function of MAX.

Table 4.4 An Example of Necessity Constraints of Function MAX

FORTRAN SOURCE		NECESSITY CONSTRAINT
	FUNCTION MAX (M, N)	
1	MAX=M	
Δ	MAX=N	$M \neq N$
Δ	MAX=ABS (M)	$M<0$
2	IF (N.GT.M) MAX=N	
Δ	IF (N.LT.M) MAX=N	$(N>M) \neq (N<M)$
Δ	IF (N.GE.M) MAX=N	$(N>M) \neq (N \geqq M)$
3	RETRUN	

Δ: mutated statement

Early experiments with necessity constraints show that test data generated for the constraints that did not involve predicates killed around 90% mutants, while the test data for the constraints that did involve predicates only killed 65% of the mutants. Although the generated test cases caused an immediate effect on the state of the mutant program, the Boolean result of the mutated predicate often was the same as that of the original program. An example of this program is shown in Table 4.5.

Table 4.5 Predicate Problem

Original statement	IF (I+K.GE.J) THEN
Mutated statement	IF (3+K.GE.J) THEN
Necessity constraint	$I \neq 3$
Test case	I=7; J=9; K=7

Although the necessity constraint $I \neq 3$ is satisfied, because I+K=14 and 3+K=10, and both of them are greater than J, the result of the predicate remains unchanged. To overcome the difficulty, Godzilla introduces the predicate constraint, $(I+K \geq J) \neq (3+K \geq J)$ in the example, to force the mutated predicates to have different result from that of the corresponding predicates in the original program.

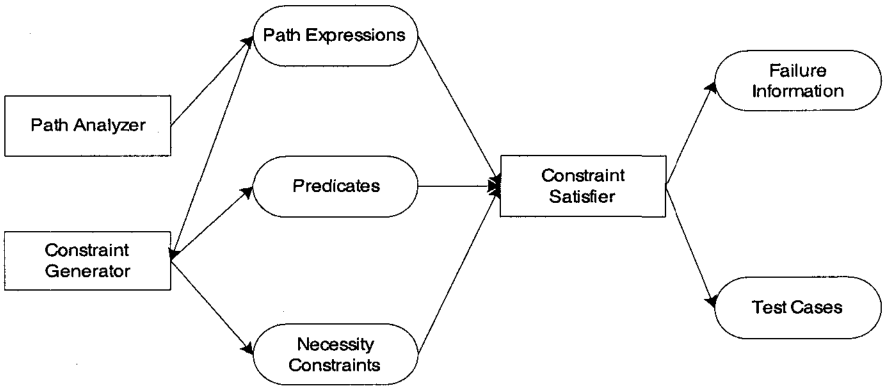

Figure 4.1 Architecture of Godzilla Automatic Test Data Generator

Figure 4.1 shows the architecture of Godzilla. The three major tools are the Path Analyzer, the Constraint Generator and the Constraint Satisfier, represented in the figure by boxes. For each statement in the original program under test, the path analyzer creates a path expression constraint such that if the path expression constraint is satisfied by the test case, the targeted statement is executed. The path expression constraint guarantees the reachability of the statement. The constraint generator constructs the predicate and the necessity constraints, which are stored in constraint tables and are passed to the constraint satisfier together with the path expression. The satisfier gets each necessity constraint in turn, finds the corresponding statement, conjoins that constraint with the appropriate path expression, and uses constraint satisfaction procedure to generate a test case to solve the conjunction. If a test case cannot be generated for some reason, the information about the failure can be supplied to the tester.

Results of using Godzilla on some small programs, presented in Offutt [1988], are listed as follows in Table 4.6 (TCs is the number of test cases; M is the number of mutants generated; K represents the number of mutants killed; E represents the number of equivalent mutants; MS is the mutation score; MS=$\frac{K}{M-E}$):

Table 4.6 Results of Godzilla on Small Programs

Program	Size	TCs	M	K	E	MS	Time
BUBBLE	10	32	339	304	35	1.00	0:22
DAYS	28	419	3016	2624	139	.95	7:02
FIND	28	58	1029	953	75	.99	2:27
GCD	55	325	5063	4747	298	.99	14:24
TRITYP	27	420	970	862	107	.99	10:53

Because of the mutation analysis basis of CBT, the test data generated has the capabilities of testing faults, and it subsumes other testing methods such as statement coverage and branch coverage. However, since CBT uses weak mutation testing, it inevitably inherits its weakness. Neither of necessity constraints and predicate constraints fully captures how an incorrect state once introduced remains incorrect until a failure is revealed as an output value. There are several reasons that can cause this to happen. Firstly, the test case is not strong enough to force the program to carry the incorrect state to the final output. Secondly, the program can be robust enough to detect the intermediate incorrect state and force the program to return to a correct state. Thirdly, the intermediate incorrect state of the mutant can be irrelevant to the final state. Moreover, CBT is lack of capability to deal with non-input variables directly, and all non-input variables have to be rewritten with symbolic evaluation in the term of input variables. With the programs getting more complicated and loops with uncertain iterations being involved, it is hard to convert all non-input variables to input variables using symbolic evaluation. Besides, even if CBT uses weak mutation testing instead of strong mutation testing to reduce the computational cost, it is still expensive to apply it on large programs. Even for the small programs like the triangle classification program, Godzilla gets 455 constraints for its constraint satisfier to work on. The constraint satisfier that Godzilla uses is domain reduction procedure, which is quite naive and inefficient. CBT also has problems handling nested expressions, arrays and loops. Although it has great results on small programs, CBT suffers from above shortcomings that prevent it from working in practical situations.

4.5. Dynamic Domain Reduction (DDR) Procedure by Offutt

A novel approach of test data generation, called dynamic domain reduction procedure, is proposed by Offutt [1999]. The dynamic domain reduction uses part of CBT, Korel's dynamic test data generation approach and symbolic evaluation. Although DDR process also works by choosing the specific path, unlike Korel's approach, it has no initial values for inputs. The values are derived from the initial input domains (range). The DDR procedure walks though the program control flow graph, generating test data along the way. Each input variable is initially given a large set of potential values (its domain), and as branches are taken in the control flow graph, the domains of the variables involved in the predicates are reduced so that appropriate predicates would be true for any assignment of values from the domain. A search process of how to reduce the domains is used. When the process is done, the remaining values for any variables' domains represent sets of test cases that will cause the traversing of the path. If any variable's domain is empty, the search process fails, which means either the path is infeasible or the procedure fails to find satisfying values to traverse the path.

One simple example to illustrate the DDR approach from Offutt [1999] is:

```
     int mid (x,y,z)
     int x,y,z;
     {
       int mid;
1      mid = z;
2      if (y<z)
       {
3        if (x<y)
4          mid = y;
5        if (x>z)
6          mid = x;
       }
7      else
       {
8        if (x>y)
9          mid = y;
10       else if (x>z)
11         mid = x;
       }
12     return (mid);
     }
```

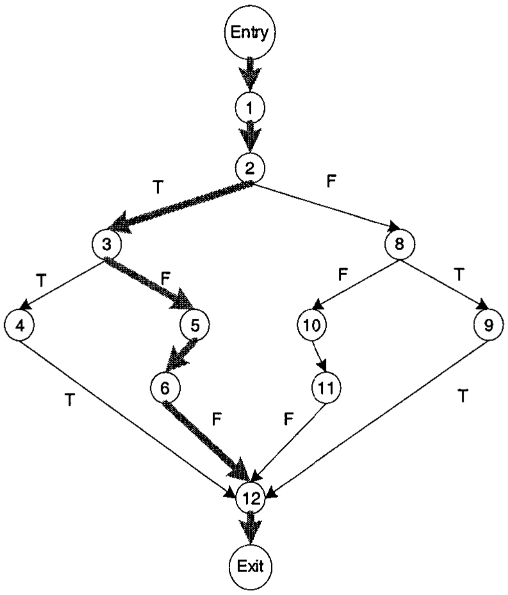

Figure 4.2 Function Mid and Its Control Flow Graph

If the path of 1-2-3-5-6-12 is chosen, input variables' domains after each constraint are as follows:

		x	y	z
1.	Start:	<-10...10>	<-10...10>	<-10...10>
2.	y < z (node 2 to 3)	<-10...10>	<-10...0>	<1...10>
3.	x ≧y (node 3 to 5)	<-5...10>	<-10...-5>	<1...10>
4.	x < z (node 5 to 6)	<-5...2>	<-10...-5>	<3...10>

The searching process for split points is crucial to the dynamic domain reduction procedure. Choosing different split points gives different results of the domain reduction procedure, and impropriate selection of split points may cause a later constraint to be infeasible. Every time the search process does the domain reduction, some of the values, which could include the solution, are kept out of the domain, so if the domain reduction procedure fails on the chosen split point, the search algorithm needs to search for a better split. During the search process, the split point is successively reevaluated using bisection, which move the split point halfway in one direction, then the other, and so on until the choice succeeds in allowing a test case to be found, or all choices have been exhausted, or a predetermined constant number of choices have been made (to avoid an infinite search).

Dynamic domain reduction improves its capability to deal with arrays by treating each element of an array as a distinct variable. It is also capable of handling paths with one loop structure by dynamically checking the loop constraint to see if another iteration of the loop is needed. The algorithm to evaluate domains for expressions, which can be very hard for other symbolic evaluation methods, is addressed in the dynamic domain reduction procedure.

Dynamic domain reduction is a more advanced approach than the original domain reduction using in CBT. However, it has several limitations: First, dynamic domain reduction procedure has only worked on small numeric operations, because it hasn't been able to handle all operations on all data objects besides numeric operations, in another word, the operation system is not yet complete. Secondly, inter-procedure issue is not mentioned at all in the algorithm, specifically, when a path goes through several procedures. Moreover, although the search algorithm for split point works for small programs like the Function Mid, when the domains for input variables become huge and continuous, it's questionable if the search process for split point can work effectively and efficiently. For example, if a program needs inputs whose type is float or even double and their domains are from the minimum to the maximum of float or double type, the search space for the split point is vast, whether the bisect algorithm works in the case is very doubtful. Finally, the dynamic domain reduction procedure works when a specific path is selected based on the program's control flow graph; however, if the dynamic domain reduction procedure is needed to work on every path of a program, this type of exhaustive testing will be computationally very expensive, and impractical for big programs with thousands of paths.

5. Test Data Generation using PDGs and GA (TDGen)

5.1. Outline

In the thesis, we propose an approach, called TDGen, which automatically generates test data using program dependence graphs and GA to test programs based on branch coverage criterion. The idea of the approach is that path selection is crucial for testing some branches, and some specific paths, which may not be easily selected and traversed by random test cases, are needed to be chosen to test those branches, therefore, to use a static program analysis to select the path or paths may help the test data generator to satisfy the desired conditions and achieve the coverage of targeted branches. TDGen uses program dependence graphs to select a path that may reach the targeted branch, and obtains constraint information of the selected path. After the constraints have been collected from the program with the assistance of program dependence graphs, a genetic algorithm is used to generate adequate test cases to satisfy all the constraints and ensure the selected path is reached and traversed. Comparing to test data generator using GA but without PDG analysis, the test data generator using both GA and PDG analysis gives GA more constraints to work with during test data generation, so it's more effective and more efficient.

Figure 5.1 shows the system architecture and dataflow of TDGen. The coverage table, similar to Figure 4.1 in Michael's approach, is established to record the branch information of the program under test, and keeps tracking whether a branch is tested or not. When the coverage table is initialized with the seed inputs, the test data generator gets next targeted untested branch from the coverage table and send query requests of the PDG analysis with regard to the target branch to the program analyzer, which takes in the source code of the program, generates a system program dependence graph including a PDG for each procedure, accepts query requests from the test data generator, and sends query results back to the data generator with either graphic user interface or application program interface. After getting query results from the program analyzer, the test data generator converts the query results to constraints. A genetic algorithm is used to satisfy the constraints and then test cases are generated. If the constraints have temporary variables in them, and the GA needs their values to evaluate fitness functions, the values of the temporary variables can be obtained by augmenting the source code of the tested program to output them. Finally, the program is run with the test cases generated by the test data generator to see if the desired result is reached. If the targeted branch or some other untested branches are traversed, the coverage table will be updated, and then new target will be selected from the coverage table for the next cycle of test data generation.

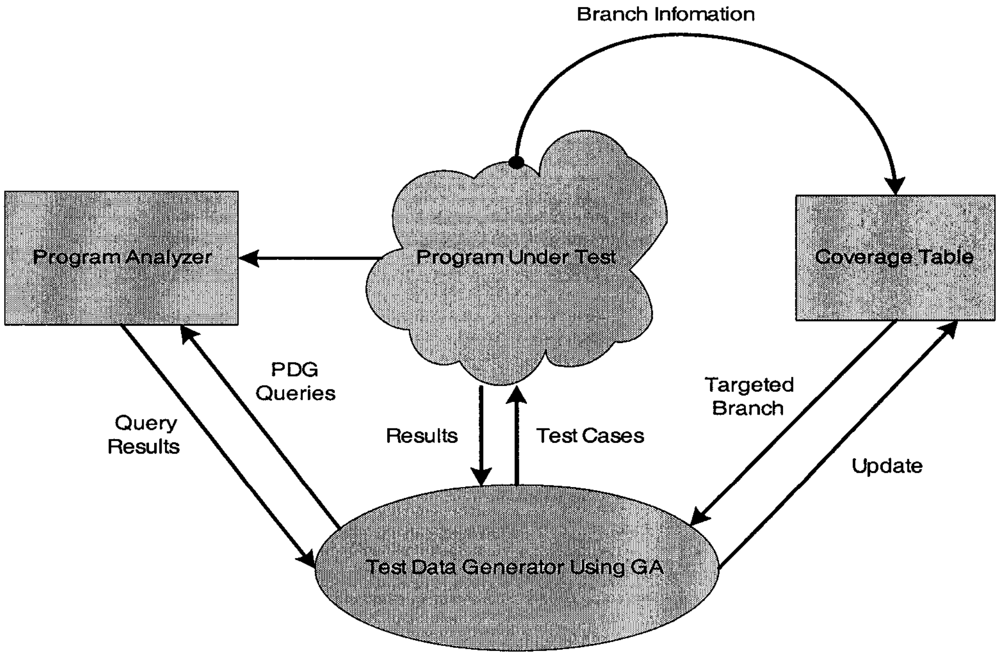

Figure 5.1 System Architecture and Dataflow Diagram

Figure 5.2 shows the algorithm of TDGen. A candidate solution (a individual or a chromosome for the genetic algorithm) is a set of test data, i.e., an array or a list of inputs for the program under test.

```
Procedure TDGen

Input:

        program: a program to be tested

Output:
```

covTable: the table to record information and testing status (tested/not tested) for each branch

testCases: set of test cases that are solution to test corresponding branches

Variable declaration:

pdgConstraint: constraint from PDG analysis

testRequirements: a table of all untested branches

curPopulation: sets of test data

nextPopulation: sets of test data

individual: a set of test data

target: one test requirement for which test data are going to be generated

maxIterations: the maximum iterations

iterationCounter: a counter to record the number of iterations

```
Begin
Create covTable;
Initialize covTable with results of random test data generation;
Create testRequirements;
For (each entry of testRequirements)
{
        target=current entry of testRequirements;
        Proceed PDG analysis for the target and convert the PDG analysis result to pdgConstraint;
        Initialize the curPopulation;
        While (iterationCounter<=maxIterations and target is not satisfied)
        {
              Compute the fitness of each individual of curPopulation;
              Select the best individual of the curPopulation to survive to be nextPopulation directly;
              Select parents from curPopulation using roulette selection scheme;
              Generate new individuals of nextPopulation from the selected parents using crossover and mutation operations;
```

```
                Execute program with each individual of the
                nextPopulation to check if there are any other untested
                branches being tested;
                Update covTable, testRequirements, testCases and
                iterationCounter;
                CurPopulation=nextPopulation;
        }
    }
    return (covTable, testCases);
    end
```

Figure 5.2 Algorithm for TDGen

5.2. Coverage Table and Priority Ranking of Branches

A triangle classification program, which is widely used in papers of automatic test data generation, is used to illustrate the strength of our approach. The triangle classification takes in three integers as three sides of a triangle, and decides which type of triangle it is to have the three sides with the length of the three integers. Four results of scalene, isosceles, equilateral and not a triangle are returned to users depending on the three integer inputs. Since the program includes some nested conditions and an enumerated data type variable, it is perfect to show the advantages of using PDGs for path selection. The source code of the triangle classification is as follow:

```
#include <stdio.h>                                                1
int triang (int i, int j, int k) {                                2
// returns one of the following:
// 1 if triangle is scalene
// 2 if triangle is isosceles
// 3 if triangle is equilateral
// 4 if not a triangle
  if ((i<=0)||(j<=0)||(k<=0))                                     3
              return 4;                          //acd            4
  int tri=0;                                                      5
  if (i==j)                                                       6
              tri+=1;                            //g              7
  if (i==k)                                                       8
              tri+=2;                            //h              9
  if (j==k)                                                       10
              tri+=3;                            //i              11
  if (tri==0) {                                  //bef            12
        if ((i+j<=k)||(j+k<=i)||(i+k<=j))                         13
```

```
            tri=4;                                         //be      14
        else tri=1;                                        //f       15
        return tri;                                                  16
        }                                                            17
  if (tri>3)                                                         18
        tri=3;                                                       19
  else if ((tri==1)&&(i+j>k))                                        20
        tri=2;                                             //j       21
  else if ((tri==2)&&(i+k>j))                                        22
        tri=2;                                             //h       23
  else if ((tri==3)&&(j+k>i))                                        24
        tri=2;                                             //j       25
  else tri=4;                                              //k       26
  return tri;                                                        27
}                                                                    28

int main (void) {                                                    29
  printf("enter 3 integers for sides of triangles\n");               30
  int a, b, c;                                                       31
  scanf ("%d %d %d", &a, &b, &c);                                    32
  int t=triang (a, b, c);                                            33
  if (t==1)                                                          34
  printf ("scalene\n");                                              35
  if (t==2)                                                          36
  printf ("isosceles\n");                                            37
  if (t==3)                                                          38
  printf ("equilateral\n");                                          39
  if (t==4)                                                          40
  printf ("not triangle\n");                                         41
}                                                                    42
```

To show how the PDG analysis works, we will focus on the procedure triang to illustrate how the path selection process proceeds. The program dependence graph of the triang procedure in the triangle classification program is given in the figure 5.3:

TDGen designed with PDG analysis and GA uses branch coverage as the test adequacy criterion. This leads to two test requirements for each predicate in the program, and either the true or false branch has to been satisfied by at least one test case. A coverage table is established, showing the predicate number, program line numbers of predicates, predicate, true/false branch and branch coverage status. Before starting to generate test data for the tested program, a seed input, usually generated randomly based on the specification or given by users, is used to execute the program under test for the first time. Generally, running the program with the seed input will have some branches tested. The result of the execution of the tested program with the seed input is recorded in the coverage table and the status of each branch is initialized.

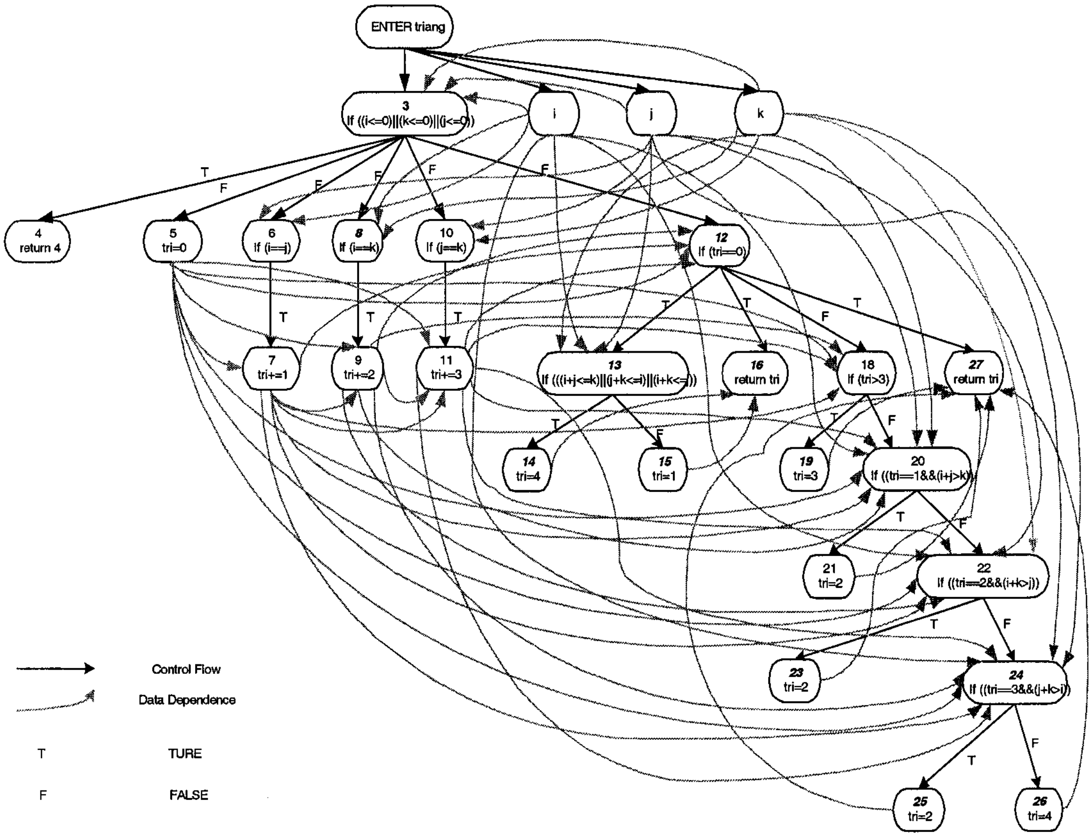

Figure 5.3: Program Dependence Graph (PDG) of Triang Procedure in Triangle Classification Program

Table 5.1 gives an example of the coverage table after the triangle program executes with the initial seed input, for example 1, 3, 4 as i, j, k. The branch coverage table is established with the information of predicate number, line number, description of predicates, true/false branch and status (traversed or not).

Table 5.1: The Coverage Table for Triangle Classification Program

#	Line	Predicate	Branch	Status (tested: 'X'; untested: '-')
1	#3	if ((i<=0)||(k<=0)||(j<=0))	True	-
			False	X

2	#6	if (i==j)	True	-
			False	X
3	#8	if (i==k)	True	-
			False	X
4	#10	if (j==k)	True	-
			False	X
5	#12	if (tri==0)	True	X
			False	-
6	#13	if (((i+j<=k)||(j+k<=i)||(i+k<=j))	True	X
			False	-
7	#18	if (tri>3)	True	-
			False	-
8	#20	if ((tri==1&&(i+j>k))	True	-
			False	-
9	#22	if ((tri==2&&(i+k>j))	True	-
			False	-
10	#24	if ((tri==3&&(j+k>i))	True	-
			False	-

After the initial coverage table has been established, the next question is which branch is our next target. Since traversing of different branches will have different contributions to the satisfaction of our selected criterion, we need to weight the importance of each branch towards branch coverage criterion. We give higher priority to those branches that their traversing causes more other branches traversed or makes other branches easier to be traversed. Some traditional dynamic test data generators deal with branches with their natural sequence in the program, and some others randomly choose one of the branches that have been reached but not tested to be the next targeted branch to work on. In our approach, we try to discriminate branches and decide their possible contributions to our software testing adequacy criterion, so as to always work on the most possible contributory branch first.

Two metrics suggested by Horwitz [2002] are used to determine the priority level of each branch. Both metrics are only for predicates. Different from Horwitz's original metrics, which use components that are statements and predicates to calculate the metrics, the two metrics in TDGen are only calculated with predicates. Horwitz's original metrics bias the predicates followed by more statements and predicates in the PDG by calculating metrics with both statements and predicates, while we don't favor predicates that have more statements in either true or false branch by calculating metrics with only predicates, because testing branches with more statements won't give us any higher branch coverage than testing branches with fewer statements.

1. Ease-of-Execution Metric:

   An estimate of the effort needed to force a predicate C of an untested branch to be executed can be computed by finding the path from a tested predicate to C that contains the fewest predicates in the program dependence graph. In the control dependence graph, each such path corresponds to one or more paths in the control-flow graph. The predicates on the program dependence graph are the "relevant" predicates on the corresponding control-flow graph paths. The Ease-of-Execution metric gives only a rough estimate of the actual effort needed to force C to execute, since in practice the predicates in a program are not independent, and it may be easier to force a predicate to evaluate to one value than to another. Nevertheless, absolute precision is not necessary, and only a reasonable correlation between the actual effort and the values of the metric is needed.

   We use the example of the triangle program with 1, 3, 4 as its inputs to calculate the Ease-of-Execution metric. From the coverage table and the program dependence graph, we know that every unreached predicate has only one path from the node Enter to itself, so there is no need to compare and decide the shortest path from the reached predicate to itself. The calculation of Ease-of-Execution metric is simple in this case (See Table 5.2 for the result).

Table 5.2: The Ease-of-Execution Metric for Each Unreached Predicate in the Triangle Program.

#	Line # (Node#)	Predicate	Ease-of-Execution Metric
7	#18	if (tri>3)	1
8	#20	if ((tri==1&&(i+j>k))	2
9	#22	if ((tri==2&&(i+k>j))	3
10	#24	if ((tri==3&&(j+k>i))	4

2. Improved-Ease-Set Metric:

Improved-Ease-Set Metric can be evaluated by calculating the total amount by which the Ease-of-Execution metrics of untested predicates are guaranteed to be lowed if a predicate P executes and evaluates to v (v represents the value that predicate P is evaluated to). This metric shows how traversing one branch impacts the Ease-of-Execution metrics of other predicates. The Improved-Ease-Set metric can be computed for each predicate P and value v by determining, for each untested predicate C reachable in the program dependence graph from P, how many predicates are on the shortest path from P to C. If the number of predicates is less than C's current Ease-of-Execution metric, then the difference in values is added to (P, v)'s Improved-Ease-Set Metric.

We still use the triangle program with inputs of 1, 3, 4 to see how to get the Improved-Ease-Set metric. Table 5.3 lists values of Improved-Ease-Set metric and nodes with lowed Ease-of-Execution metric in the bracket for each predicated that has not been reached.

Table 5.3 Improved-Ease-Set Metric for Each Untested Branch in the Triangle Program.

#	Line # (node #)	Predicate	Branch	Lowed Ease Metric	Improved-Ease-Set Metric
1	#3	if ((i<=0)||(k<=0)||(j<=0))	True	None	0
			False	X	X
2	#6	if (i==j)	True	None	0
			False	X	X
3	#8	if (i==k)	True	None	0
			False	X	X
4	#10	if (j==k)	True	None	0
			False	X	X
5	#12	if (tri==0)	True	X	X
			False	18 (-1), 20(-1), 22(-1), 24(-1)	4
6	#13	if (((i+j<=k)||(j+k<=i)||(i+k<=j))	True	X	X
			False	None	0
7	#18	if (tri>3)	True	20(-1), 22(-1), 24(-1)	3
			False	20(-2), 22(-2), 24(-2)	6
8	#20	if ((tri==1&&(i+j>k))	True	22(-2), 24(-2)	4
			False	22(-3), 24(-3)	6
9	#22	if ((tri==2&&(i+k>j))	True	24(-3)	3
			False	24(-4)	4
10	#24	if ((tri==3&&(j+k>i))	True	None	0
			False	None	0

The Ease-of-Execution metric gives high values to branches that are closer to the ENTER node in program dependence graph, and the Improved-Ease-Set metric gives high values to the branches that may have more contribution to the goal of branch coverage criterion. The two metrics computed for each untested predicate or branch can be used to determine an ordering by which TDGen can choose which branch to focus on next. TDGen sorts all branches using Ease-of-Execution metric as the primary key (from low to high), and Improved-Ease-Set metric as the second key (from high to low). The intuition of using Ease-of-Execution metric is that predicates having lower values of Ease-of-Execution metric are closer to the ENTER node in PDG, and it is relatively easier for the data generator to find inputs to force these predicates to execute. The Improved-Ease-Set metric makes the data generator bias those more profitable branches, because their execution is likely to make more untested components to be executed. For instance, traversing of the false branch of predicate #7, *if (tri>3)* on line #18, seems more fruitful than the execution of the true branch, because false branch may cause the execution of predicates 22 and 24.

Therefore, after triangle classification program is run with the seed input of 1, 3, 4, and the coverage table is initialized, all untested branches in the coverage table are sorted with Ease-of-Execution first and then Improved-Ease-Set. The values of the two metrics and the sorted untested branches are in Table 5.4.

Table 5.4: Untested branches Sorted with Ease-of-Execution as Primary Key (Low to High) and Improved-Ease-Set as Secondary Key (high to low) of the Triangle Program

#	Line	Predicate	Branch	Ease-of-Execution Metric	Improved-Ease-Set Metric
5	#12	if (tri==0)	False	0	4
1	#3	if ((i<=0)||(k<=0)||(j<=0))	True	0	0
2	#6	if (i==j)	True	0	0
3	#8	if (i==k)	True	0	0

4	#10	if (j==k)	True	0	0
6	#13	if (((i+j<=k)||(j+k<=i)||(i+k<=j))	False	0	0
7	#18	if (tri>3)	False	1	6
			True	1	3
8	#20	if ((tri==1&&(i+j>k))	False	2	6
			True	2	4
9	#22	if ((tri==2&&(i+k>j))	False	3	4
			True	3	3
10	#24	if ((tri==3&&(j+k>i))	True	4	0
			False	4	0

In table 5.4, the false branch of #5 predicate is listed as the first target for the test data generator to work on. Although it has the same value of Ease-of-Execution as other untested branches of predicate #3, 6, 8, 10 and 13, traversing the false branch of predicate #5 has more contribution to our branch coverage criterion according to its higher value of Improved-Ease-Set metric. It is showed in the PDG of the procedure *triang* that when the false branch of predicate #5 is traversed, predicate #18, #20, #22 and #24 are closer to the closest reached node in the PDG, and may be easier for the test data generator to test them with less effort. It is indicated in the table that all false branches of predicate #18, #20, #22 and #24 have higher priority over corresponding true branches in that each false branch leads to lowing the values of other predicates' Ease-of-Execution metric, while every true branch doesn't.

5.3. Path Selection Using PDG

Since the algorithm of GADGET in Michael [2001] has been explained in the last chapter, comparison between TDGen and GADGET will be used in this section to show the advantages of TDGen. Both approaches use GA, and the difference is that TDGen uses program dependence analysis of the tested program to guide GA to work more effectively.

The triangle classification program is an excellent example for illustrating most test data generation algorithms and provides some challenges to different approaches, because of the number of predicates, including nested predicates that exist in such a small program. The temporary variable *tri* is an enumerated type, which is a hard job for most GA approaches. The search space of the three integers is huge, from negative minimum integer (-2,147,483,647) to positive maximum integer (2,147,483,647). The large search space decreases the chance for a random search approach to coincidentally satisfy many of the conditions, and by the mean time it helps to show the evolving procedure of the genetic search algorithm.

First, let us take a look at some results from GADGET on the triangle program. Table 5.5 provides the test cases that succeed traversing some of the branches in the program by GADGET reported in Michael [2001]. Key a to i are marked in the program source code displayed in Section 5.2.

Table 5.5: Test Cases Generated by GADGET for Triangle Classification Program

Number of Runs	Key	Integer 1	Integer 2	Integer 3
2	a	1680498885	1961702355	-1490056820
3	b	1293470477	1898197634	465181194
4	c	-1201922928	1041962067	280365949
6	d	841354299	-1802686561	-209782529
20	e	1056804119	660913846	1617709752
117	f	719320455	507534636	574028437
5311	g	743820356	743820356	1826109949
10751	h	999699718	584551117	999699718
16800	i	799340978	1321708382	1321708382

By analyzing the test cases with the triangle program, we can see that there are 3 conditions that GADGET hasn't satisfied in the procedure *triang*. They are:

$tri>3$	in	*if(tri>3)*	*on line 18;*
$i+j>k$	in	*else if((tri==1)&&(i+j>k))*	*on line 20;*
$j+k>i$	in	*else if((tri==3)&&(j+k>i))*	*on line 24;*

The reason why GADGET doesn't satisfy the last condition on line 24 with the input set of 799340978, 1321708382 and 1321708382, which is supposed to reach and satisfy $j+k>i$, is that GADGET doesn't take integer overflow into its consideration. The range for i, j, k is from (-2,147,483,647) to (2,147,483,647), so when the sum of j and k is larger than the maximum for integer type, the memory overflow causes the sum of j and k to be a negative value which is smaller than i. This is just a little overlook of GADGET and it is easily fixed in TDGen.

Since GADGET only uses the conditions of the targeted predicate as the constraint for the GA to work on, it's easy for the GA to find valid inputs to satisfy conditions like $i=j$, $j=k$, $i=j$ and $i+j<k \mid\mid j+k<i \mid\mid i+k<j$, which have only inputs and are easy to reach. However, when a specific path is required to be selected to reach a predicate, GADGET has trouble reaching and satisfying it. In addition, the overflow situation lies, so the only luck that GADGET got in the triangle program is

$i+k>j$	in	*else if((tri==2)&&(i+k>j))*	*on line 22;*

The condition of $i+k>j$ is satisfied as a by-product of GADGET satisfying the condition

$i==k$	in	*if(i==k)*	*on line 8;*

When GADGET satisfies the constraint of $i=k$, the condition of $i+k>j$ might be satisfied with serendipity. Because the GA starts with a random seed input, the new input that GADGET generates when attempting to satisfy *i==k* on line 8 could happen to be, for example,

999699718, 1984551117, 999699718 (i+k<j),

instead of the fortunate

999699718, 584551117, 999699718 (i+k>j)

Because the input with i+k<j also reaches line 22, when GADGET targets *i+k>j* on line 22 later, it may fail in the GA process to evolve the input with i+k<j to the new input with i+k>j, just like it's failure with another input

743820356, 743820356, 1826109949 (i+j<k)

for

i+j>k in *else if((tri==1)&&(i+j>k))* *on line 20;*

The reason why GADGET fails to evolve the input in which i+j<k to the input in which i+j>k is that it doesn't consider the necessary path that the input should traverse to reach the condition as well as satisfy it. The algorithm has no mechanism to analysis data dependence and it considers each predicate separately: when the GA targets i+j>k, it doesn't consider that fact that *tri=1* is the result of *i=j*. Therefore, the fitness functions that GADGET uses for the condition might be $\begin{cases} k-i+j & if k>i+j \\ 0 & otherwise \end{cases}$ and $\begin{cases} |tri-1| & if tri \neq 1 \\ 0 & otherwise \end{cases}$. When the GA does the crossover and mutation operation on the inputs that satisfy *tri=1* and reach the condition *i+j>k*, and change the values of i and j to improve the fitness value. If the new inputs cannot reach the condition because of i ≠ j, a low fitness value is given to the new input, even if i is very close to j, because neither the first nor the second fitness function is able to help the algorithm to make the values of i and j go closer to each other. Since the evolvement process of the GA is interfered by the fact that the condition is reached or not, GA is incapable of distinguishing a better set of inputs from a worse set of inputs if neither of them reaches the predicate of the targeted branch, and GA may keep abandoning new inputs unable to reach the target condition, thus finally stops because of no improvement within a certain amount of generations. If we can think of

i+j>k in *else if((tri==1)&&(i+j>k))* *on line 20;*

and because *tri=1*, we add

i=j in *if(i=j)* *on line 6;*

then we can find an adequate test input by minimizing the function $\Im_1+\Im_2$, where

$$\Im_1 \begin{cases} k-i+j & if k > i+j \\ 0 & otherwise \end{cases}$$

$$\Im_2 \begin{cases} |i-j| & if i <> j \\ 0 & otherwise \end{cases}$$

Therefore, a specific path reaching and satisfying the conditions in the predicates on line 6 and line 20 should be chosen, and then the conditions in both predicates can be converted to a function to transform it to be a minimization problem.

As for

tri>3 in *if(tri>3)* *on line 18,*

GADGET fails to satisfy *tri>3*, because each of the conditions

i=j in *if(i=j)* *on line 6;*

i=k in *if(i=k)* *on line 8;*

j=k in *if(i=j)* *on line 10;*

is treated separately in the coverage table and there is no mechanism to connect the local variable *tri* on line 18 to above conditions. Due to GADGET having no knowledge of the structure of the program, there is no mechanism to allow it to derive the relationship between the local variable *tri* and the inputs of i, j and k. Suppose that GADGET has the inputs that satisfy the program reaching line 18 (say, i=j<>k), and uses the fitness function $\Im = \begin{cases} 3 - tri & if\, tri < 3 \\ 0 & otherwise \end{cases}$, the chance that GA makes i=j=k is extremely slim, due to $\Im$ is equal to 2 for the whole duration of the GA process. The search is effectively equivalent to a random search. If we can use program dependence analysis to derive the fact that the value of *tri* is changed at lines 7, 9, 11 and push the test data generator to produce input that choose a path to go through lines 7, 9, 11, the chance to satisfy the desired condition *tri>3* on line 18 is much bigger.

Some branches can be traversed by GADGET in Michael [2001] described in the previous chapter. Michael's approach fails in some cases that are difficult for GA to score, especially when the program gets bigger and complicated, since it has no assistance from program dependence analysis. Michael's approach depends completely on the power of GA and all it has is the information of the targeted predicates and if they have been reached or not. PDG analysis is helpful to resolve this shortcoming by providing some extra program information for GAs to work successfully. Because static analysis like PDG analysis is relatively computational expensive, comparing to dynamic approaches without static analysis, there is always no hurt to apply PDG analysis only after using dynamic approach to have some branches tested. To further reduce the computational cost, we apply control dependence analysis first, and data dependence analysis is added only when GA and GA with control dependence fail to have the targeted branches tested. The idea of using data dependence is to find out all possible related paths in the PDG with regard to the target predicate, and then a GA is used with data dependence to generate test cases to traverse each related paths that reach the targeted predicate and traverse the targeted branch. With the programs getting bigger, the use of GA to generate test cases to traverse all the possible paths could be very expensive. We only need to do it on those tough branches on which GA and GA with control dependence analysis are unsuccessful.

Let's consider the triangle program once again to see how path selection mechanism works in the proposed approach, and take those failed cases of

GADGET for TDGen to do program dependence analysis to find the right path to reach and satisfy the untested branches. From the proposed solution by GADGET, one untested branch that GADGET is incapable to traverse is the true branch of

else if((tri==1)&&(i+j>k)) *on line 20;*

From the PDG of the procedure triang in the triangle program (See figure 5.4), we can see that node 20 is control dependent on nodes 3(F), 12(F), and 18(F), i.e., to reach node 20, the FALSE branches of predicates 3, 12 and 18 must be taken, although there are still several paths between node 3 and node 12 on the control flow graph. To take the true branch of node 20, the conditions in the predicates are also needed to be satisfied, so *tri=1* and *i+j>k*. With control dependence analysis and the condition information from the targeted branch, the constraint that we can get from the PDG is:

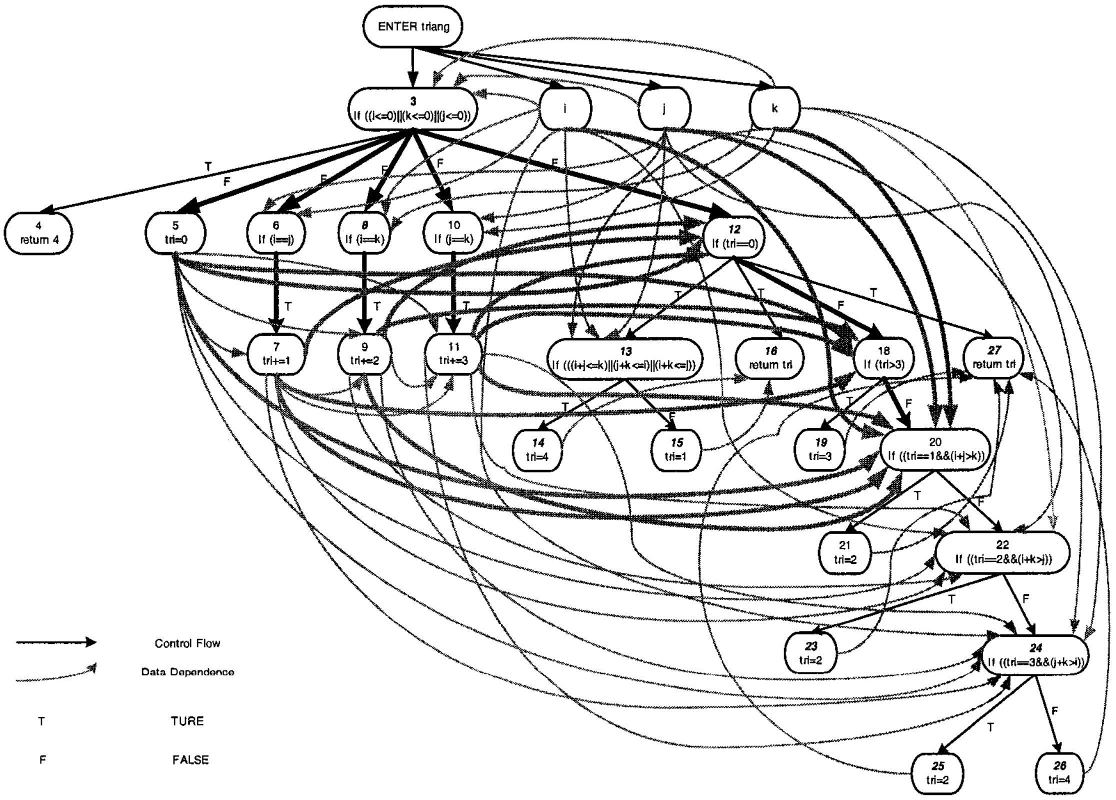

Figure 5.4: Control Dependence and Data Dependence Analysis for Node 20

(i>0 && j>0 && k>0) && (tri<>0) && (tri>3)

AND

(tri=1) && (i+j>k)

The constraint after deleting overlapped conditions is:

(i>0 && j>0 && k>0)&& (tri=1) && (i+j>k)

With the constraint that we get after the control dependence analysis, one more constraint (i>0 && j>0 && k>0) is obtained than Michael's approach in this case. This constraint can keep GA from wasting time and resource to generate test cases with i<=0||j<=0||k<=0.

When GA fail to generate test cases to test the targeted branch with the constraints obtained from control dependence analysis, it means that GA fail to find the proper path to reach the targeted predicate or fail to traverse either the true or the false branch. Therefore, Data dependence analysis needed to be considered to find all related paths that reach the targeted predicated and may traverse the targeted branch, and GA needs to try to generate test cases to traverse each of the paths, so as to fulfill the goal of testing the targeted branch.

We need to do data dependence analysis on each variable in the constraint that we have got from the control dependence analysis. In the previous case, since i, j and k in the condition (i+j>k) and condition (i>0 && j>0 && k>0) are only data dependent on input variables, no further data dependence analysis is necessary. For the condition of *tri=1*, we need to undertake data dependence analysis to see where the variable *tri* is manipulated. This allows us to generate new inputs to try to traverse the paths that may change the value of *tri.* This approach shows that *tri* at node 20 is data dependent on nodes 7, 9 and 11, i.e., when the value of *tri* at node 7 or 9 or 11 is changed, it will propagate into value of *tri* at node 20. In the PDG, node 7, 9 and 11 are three assignment statements of *tri.* Only those assignments of *tri* that are control dependent on nodes other than the entry node may change the value at node 20, so assignments of *tri* that are only control dependent on the entry node are not able to change the value of *tri* at node 20 no matter which path is chosen. The nodes that assignments of *tri* are control dependent on are considered instead of nodes of the assignment nodes themselves. For instance, for nodes 7, 9 and 11, nodes 6, 8 and 10 are considered instead. The different execution combinations of the assignments of *tri* that *tri* at node 20 is data dependent on change the value of the *tri* at node 20. Each combination is a possible path to reach the targeted branch and change the value of *tri*, thus by trying all different combinations of whether predicates 6, 8 or 10 is satisfied, a combination or path can be found to make the value of *tri* at node 20 change from *tri=0* to *tri=1*. The permutation operation is used for the combination of the different predicates in the data dependent analysis. For each of predicates 6, 8 and 10, both control dependence analysis and data dependence analysis need to be proceeded with regard to node 20. (See figure 5.5 for the constraint information obtained from control and data dependence analysis)

The control dependence analysis (connected by bold black lines) and the analysis of the predicates themselves (connected by thin black lines) use AND operations, and the data dependence analysis (connected by bold grey lines) uses permutation operations (∈). The constraint that test cases have to satisfy to test the FALSE branch of node 20 is

i+j>k

AND

tri=1

AND

(i>0 && j>0 && k>0&&i=j) (i>0 && j>0&&k>0&&i=k) (i>0 && j>0 && k>0 && j=k)

AND

(i>0&&j>0&&k>0)

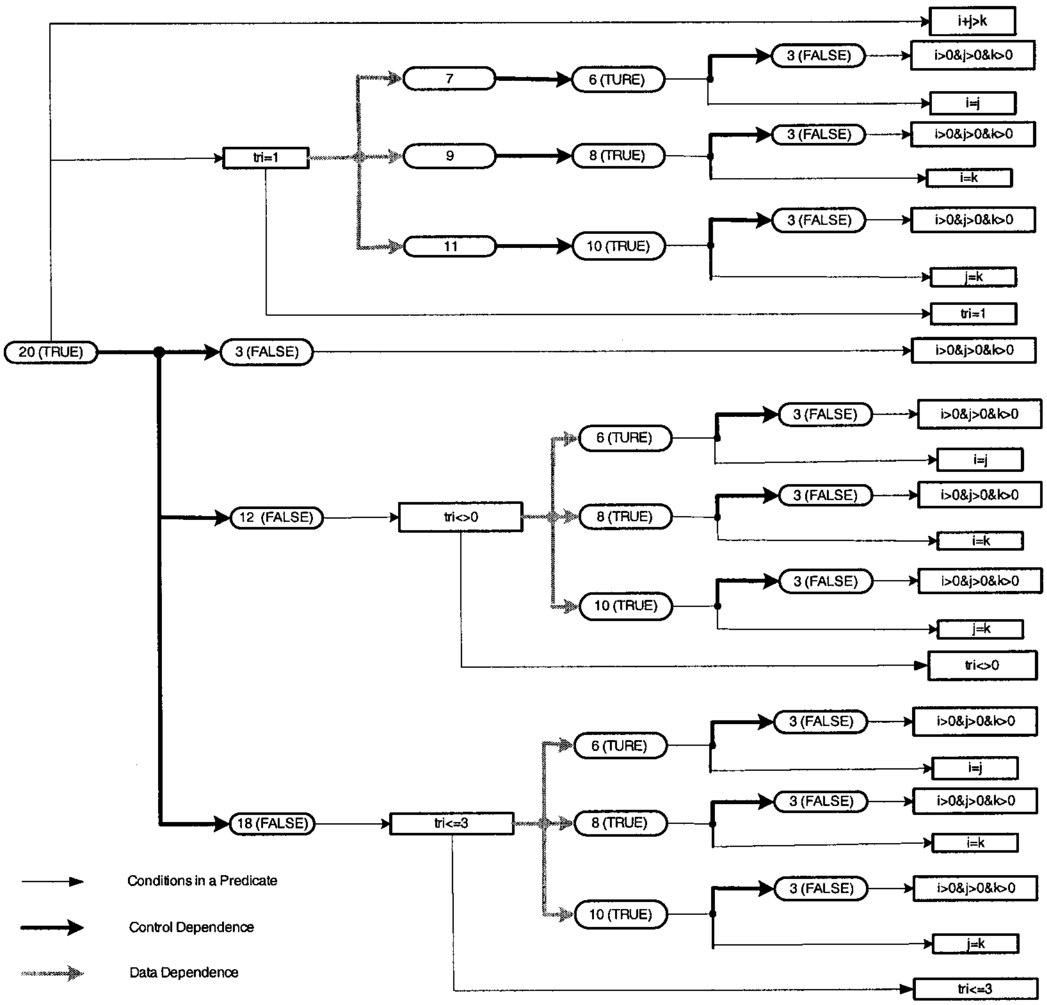

Figure 5.5: Constraints Information from Control and Data Dependence Analysis

AND

tri<>0

AND

(i>0 && j>0 && k>0 && i=j) (i>0 && j>0&&k>0&&i=k) (i>0 && j>0 && k>0 && j=k)

AND

tri<=3

AND

(i>0 && j>0 && k>0 && i=j)   (i>0 && j>0&&k>0&&i=k)   (i>0 && j>0 && k>0 && j=k)

Finally, by deleting the duplicate constraints, the ultimate constraint for the GA to generate inputs is

i+j>k

AND

tri=1

AND

(i>0 && j>0 && k>0&&i=j)   (i>0 && j>0 && k>0 && i=k)   (i>0 && j>0 && k>0 && j=k)

AND

(i>0&&j>0&&k>0)

and they can be further simplified to

i+j>k && tri=1 && (i=j   i=k   j=k) && (i>0 && j>0 && k>0)

Since the different permutations of *i=j, i=k, j=k* manipulate the value of *tri* on node 20, and each permutation is the possible path to reach node 20 and satisfy the condition of *tri=1* in node 20. The eight permutations of *i=j, i=k, j=k* are as follows:

I. i<>j && i<>k && j<>k

II. i==j && i<>k &&j <>k

III. i<>j && i==k && j<>k

IV. i<>j && i<>k && j==k

V. i==j && i==k && j<>k

VI. i==j && i<>k && j==k

VII. i<>j && i==k && j==k

VIII. i==j && i==k &&j ==k

TDGen first randomly chooses one of the above eight permutations and construct a constraint with i+j>k && tri=1&& (i>0 && j>0 && k>0), then GA tries to find test cases to satisfy the constraint to ensure the selected path is traversed. If GA fails to generate at least one test case to satisfy the constraint, another permutation will be selected to construct another constraint and GA will try again to find a test case to satisfy it. The process will keep going until all permutations or related paths are tried. TDGen will give up working on the targeted branch, when the GA fails to find a test case to traverse one of the paths obtained from the control and data dependence analysis and the targeted branch. The reason that the algorithm gives up testing targeted branch is that either the targeted branch is unreachable or non-traversable or the GA cannot find a test case to satisfy the constraint from PDG analysis, sequentially, cannot find a test case to reach and traverse the targeted branch.

The control and data dependence analysis in this section is only for illustrating the proposed approach, since it's practically impossible to manually generate a PDG to analyze the control and data dependence for a practical real world program. CodeSurfer is used to get results of control and data dependence analysis, and it makes PDG analysis easier and applicable on large programs.

5.4. The Genetic Algorithm in TDGen

When a path is selected and the constraint is constructed, a fitness function needs to be constructed for the GA to evaluate the fitness of each test case it generates.

Table 5.6 shows the fitness functions corresponding to basic relational operations in source code.

Based on the rule, For example, the constraint of permutation II is:

i+j>k && tri=1 && (i==j && i<>k &&j <>k) && (i>0 && j>0 && k>0)

Table 5.6: Rules to Build Fitness Function for Basic Relational Operations

Example	Fitness function (p is a preset penalty) value)
If (a)…	$\Im = \begin{cases} 0 & TRUE \\ p & FALSE \end{cases}$
If (a=b)	$\Im = \begin{cases} 0 & a = b \\ abs(a-b)+p & a \neq b \end{cases}$
If (a<>b)	$\Im = \begin{cases} 0 & a \neq b \\ p & a = b \end{cases}$
If (a<b)	$\Im = \begin{cases} 0 & a < b \\ (a-b)+p & a \geq b \end{cases}$
If (a<=b)	$\Im = \begin{cases} 0 & a \leq b \\ (a-b)+p & a > b \end{cases}$

The maximum operator is used for AND operations, and minimum operator is used for OR operations. The fitness function for permutation II is:

$$\Im = \text{maximum} \ (\begin{cases} 0 & i+j>k \\ (i+j-k)+p & otherwise \end{cases}, \begin{cases} 0 & tri=1 \\ abs(tri-1)+p & otherwise \end{cases},$$

$$\begin{cases} 0 & i=j \\ abs(i-j)+p & i \neq j \end{cases}, \begin{cases} 0 & i <> k \\ p & otherwise \end{cases}, \begin{cases} 0 & j <> k \\ p & otherwise \end{cases}, \begin{cases} 0 & i>0 \\ -i+p & otherwise \end{cases},$$

$$\begin{cases} 0 & j>0 \\ -j+p & otherwise \end{cases}, \begin{cases} 0 & k>0 \\ -k+p & otherwise \end{cases})$$

After the test data generator initializes the first population for the genetic algorithm, each test case is evaluated according to the fitness function, and given a fitness value. If the constraint involves temporary variables, for example, *tri* in the case of permutation II, the program needs to be run with each generated test case to get the value of the *tri* at the line #20 of the triangle classification program. If the designated program location where the temporary variables lie has not been reached by the test case, a special penalty is given as the fitness value for the part of the fitness function with the temporary variables, and in the case of permutation II, the fitness function for "tri=1" is changed to:

$$\begin{cases} 0 & tri = 1 \\ abs(tri - 1) + p & tri <> 1 \\ p' & tri \quad unreached \end{cases}$$

p' is a penalty bigger than p, because we believe that test cases that reach the predicate being evaluated should be fitter than test cases unable to reach the predicate.

Tournament selection with replacement between two individuals is used as the selection scheme in our GA. To generate a parent, the GA takes out two individuals randomly with uniform possibility from the current population pool, compares the fitness values of the two selected individuals, chooses the one with the better fitness value as a parent, and puts the two individuals back to the population pool for the next selection of a parent.

For the genetic operations, real coding is used for both crossover and mutation operations. Not every pair of parents do crossover to generate a new pair of offspring of the next generation. A crossover and a mutation probability are set up so that each pair of parents just has the presetting probability to do crossover, and each of the offspring after the crossover operations only mutates with the presetting probability. A random number between 0 and 1 is generated to decide if a crossover operation or a mutation operation is needed. If the random number is less than the presetting crossover probability or mutation probability, a crossover or mutation operation needs to be done, otherwise, the crossover or mutation operation is skipped. If crossover is skipped, both of the parents will be kept intact, and passed on to the mutation procedure. Single crossover is used for crossover operator. If the mutation is needed to do on one individual, one of the three mutation operators, uniform random mutation, non-uniform random mutation and Mühlenbein's mutation, is randomly chosen as the mutation operator applied on the individual. The reason of randomly selecting mutation operator is to introduce more randomness of mutation operations, and to reduce the shortcomings of different mutation operators when they are used separately.

In the new generation, the best individual of the past generation is always kept and passed to the new generation. By doing this, we want to make sure the evolution process won't degenerate from one generation to the next generation, and the best individual won't be lost by any chance.

When the new generation is produced, the fitness evaluation procedure will be done again to evaluate each of the individuals in the new generation. The GA keeps doing evaluation, crossover and mutation until the fitness function reach its target minimum value zero, i.e., the constraint is satisfied by at least one of the individual in the current generation. The test case or test cases that satisfy the constraint are used to run the tested program to verify if the targeted branch is traversed or not. When it is confirmed that the targeted branch is tested, the coverage table is updated and the test cases are recorded. The GA gives up if a test case cannot be found to satisfy the constraint after a certain amount of generations, or the values of fitness function stops getting smaller in a certain amount of generations.

During the GA process, every time a new generation is generated, the program is run with each individual test case to see if the branch coverage has any improvement. The reason to do this is that some untested branches may be traversed coincidentally when the GA works on other branches. In fact, this kind of coincidences can happen a lot in the GA evolving process. In the triangle program, it's easy to get three integers constructing an isosceles triangle when the GA is trying to generate i, j, k to satisfy the requirement of getting an equilateral triangle. When an untested branch is traversed, no matter whether it is the targeted untested branch or not, the coverage table is updated with the new achievement and the test case is recorded.

After testing the targeted branch is fulfilled or some other untested branches are traversed, another untested branch will be selected as the next target according to the two selection metrics in 5.2. The control dependence and data dependence analysis are repeated, and new GA procedure is used to satisfy the obtained constraint. The recursive process terminates when all the branches in the coverage table are tested, or a preset coverage percentage is reached, or no more branch has been left for testing.

5.5. Discussion of Computational Cost

Since the TDGen is a heuristic process, it is hard for us to predict how many iterations there will be for each chosen requirement. However, there are several facts that influence how complex the proposed approach could be.

In the real world, the programs under test are larger and more complicated than triangle classification program. The control dependence and data dependence analysis can become very complicated. In addition, the constraint information that the algorithm gets from the PDG analysis may be extremely complex. If there are many OR operators in the predicate for the targeted branch, or the data dependence analysis produces a lot of permutation operations in final constraints to guide the GA, the entire process of the data generator to find a proper input to reach and traverse the targeted branch will be very difficult and computational expensive. Suppose the number of permutation operations in the final constraints is N, then there will 2^N possibilities that the date generator has to try to find the solution constraints, and the corresponding test cases. Although if the data generator is given enough time to do the exhausted search, i.e. try all 2^N possible combinations of all constraints, and the data generator is capable to find the solution, the cost will be high when N is huge. For instance, if there is 20 permutation operations in the constraint we get from the PDG analysis, in the worst case of exhausted search, the data generator has to get 2^{20}=1048576 different constraints and try to generate millions of test cases to satisfy 1048576 constraints one by one, and the tested program needs to be run for each generated test case.

From the triangle classification example which has a relatively complicated nesting structure in a 42 line program, we can see that although the constraint that we get from the PDG analysis is complex at first, it becomes simpler after the redundant conditions have been deleted. We anticipate that in most practical programs, this is the typical case that the constraint could be simplified to make the GA process easier. The probability of a program to have very complicated data dependence among local variables is not very often. Therefore, we believe that TDGen works for most practical programs with reasonable complexity of data dependence.

5.6. Issues Regarding Loops and Arrays

Loops are always tough for automatic test data generation researches. No method can handle them perfectly, either it is ignored in dynamic test data generators or it is handled in an inefficient way, which makes it impossible to apply on big programs in the real world, such as unrolling in the static test data generation. In TDGen, only false branch and the first iteration of true branch are considered, so loops are treated like they are condition structures, and predicates in loops are handled as if they are predicates in condition structures. In this way, we avoid trying to unroll the loops, while we still partially consider them in the program dependence analysis.

In program dependence analysis, each element of an array is treated like it is a distinct variable, and values of elements of non-input arrays can be obtained by augmenting the source code to output the values, like the way temporary variables are handled. When arrays involved in data dependence analysis, they could make the results very complicated. However, since data dependence is applied only when GA and GA with control dependence fail, the chance of TDGen to solve arrays is bigger than each of the three single methods.

5.7. Results on the Triangle Program

In Section 5.3, it is discussed that how GADGET fails to reach the predicates on line 18, 20 and 24, and test the true branches of them. In this section we use

TDGen to work on each of the three branches that GADGET was unable to test. To test the true branch of the predicate, if *tri>3*on line 18, TDGen needs to generate three equivalent integers to make up of an equilateral triangle. As for the true branch of *if ((tri==1)&&(i+j>k))* on line 20 and the true branch of *if ((tri==3)&&(j+k>i))* on line 24, two sets of three integers with i=j<>k and i<>j=k respectively are needed to construct isosceles triangles.

For each predicate, a PDG graph (if it is not provided in Section 5.3), the constraint derived using PDG analysis and the result of two trials using the GA with corresponding fitness functions is presented:

1. *if (tri>3)* *on line 18;*

The PDG is showed in Figure 5.6 and the constraint derived from the PDG analysis is:

(i=j   i=k   j=k) && (i>0 && j>0 && k>0) &&tri>3

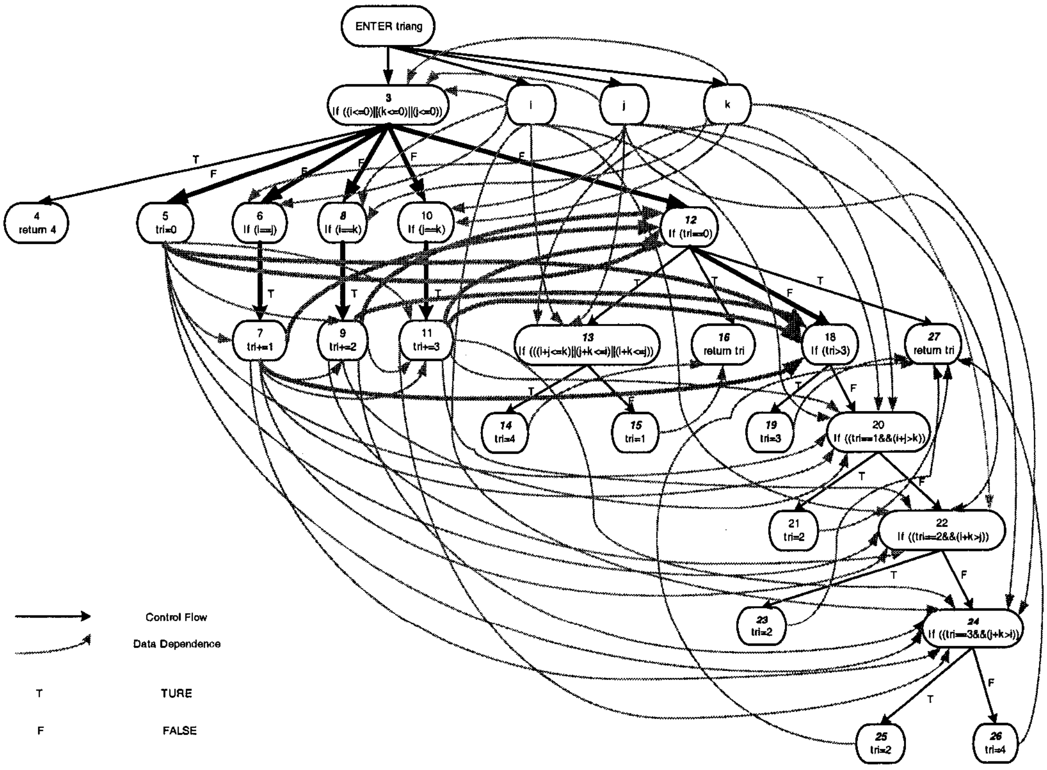

Figure 5.6: Control Flow and Data Dependence Analysis for Node 18

When i=j=k is selected, the condition of *tri>3* is satisfied and the true branch of node 18 is traversed.

For i=j=k && (i>0 && j>0 && k>0), the fitness function is

$$\Im \;=\; \text{maximum} \;\; (\begin{cases} 0 & i=j \\ abs(i-j)+p & i \neq j \end{cases}, \begin{cases} 0 & i=k \\ abs(i-k)+p & i \neq k \end{cases},$$

$$\begin{cases} 0 & j=k \\ abs(j-k)+p & j \neq k \end{cases}, \begin{cases} 0 & i>0 \\ -i+p & otherwise \end{cases}, \begin{cases} 0 & j>0 \\ -j+p & otherwise \end{cases},$$

$$\begin{cases} 0 & k>0 \\ -k+p & otherwise \end{cases}, \begin{cases} 0 & tri>3 \\ 3-tri+p & otherwise \end{cases})$$

Trial #1: 827077106 827077106 827077106 in Generation #309

Trial #2: 1190496163 1190496163 1190496163 in Generation #179

2. *if((tri==1)&&(i+j>k))* *on line 20;*

As the demonstration in last section shows, the constraints are

i+j>k && (i=j i=k j=k) && (i>0 && j>0 && k>0) && tri=1

By selecting the combination of i=j, i<>k, j<>k, the condition of *(tri==1) && (i+j>k)* is satisfied and the true branch of the node 20 is traversed.

$$\Im \;=\; \text{maximum} \;\; (\begin{cases} 0 & i+j>k \\ (k-i-j)+p & otherwise \end{cases}, \begin{cases} 0 & i=j \\ abs(i-j)+p & i \neq j \end{cases},$$

$$\begin{cases} 0 & i <> k \\ p & otherwise \end{cases}, \begin{cases} 0 & j <> k \\ p & otherwise \end{cases}, \begin{cases} 0 & i>0 \\ -i+p & otherwise \end{cases}, \begin{cases} 0 & j>0 \\ -j+p & otherwise \end{cases},$$

$$\begin{cases} 0 & k>0 \\ -k+p & otherwise \end{cases}, \begin{cases} 0 & tri=1 \\ abs(tri-1)+p & otherwise \end{cases})$$

Trial #1: 296576321 296576321 71684961 in Generation #158

Trial #2: 342425063 342425063 606335952 in Generation #161

3. *if((tri==3)&&(j+k>i))* *on line 24;*

Figure 5.7 is the PDG showing the control and data dependence analysis for node 24 and the constraints derived from the PDG analysis are:

j+k>i && (i=j i=k j=k) && (i>0 && j>0 && k>0) && ((tri<>1)|| (i+j<k)) && ((tri <>2)|| (i+k<j))

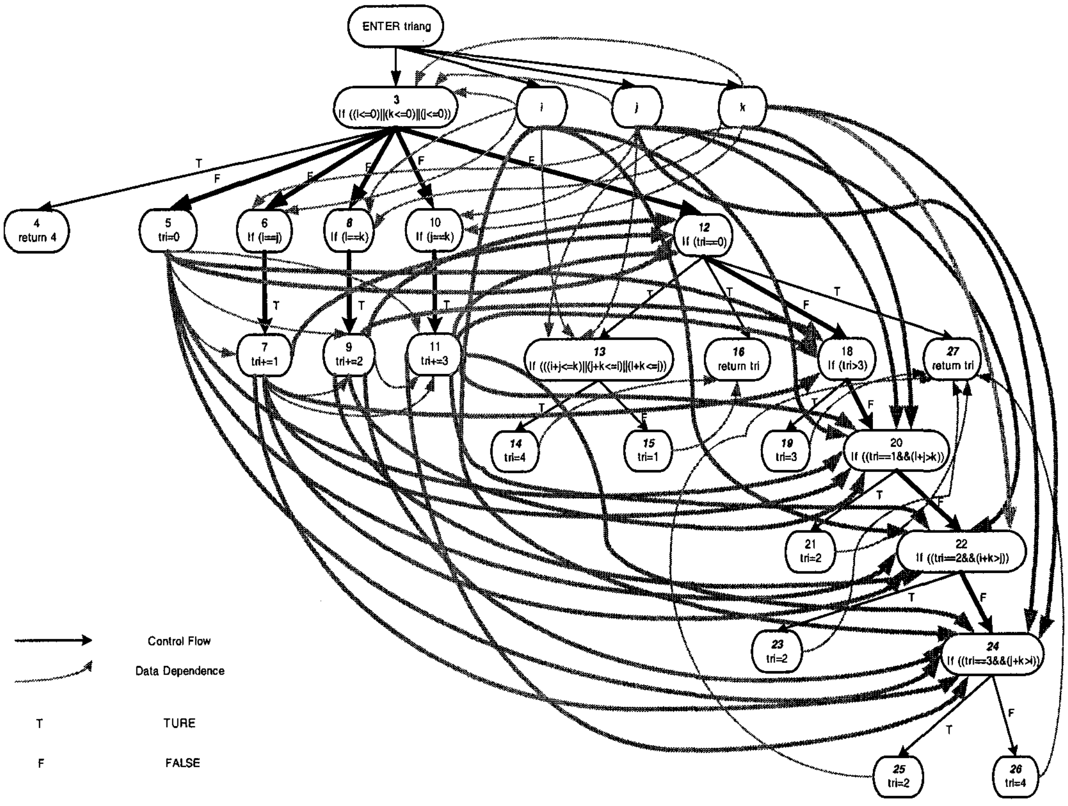

Figure 5.7: Control Flow and Data Dependence Analysis for Node 24

By selecting the combination of i<>j, i<>k, j=k, the condition of *(tri==3) && (j+k>i)* is satisfied and the true branch of node 24 is traversed.

$$\Im = \text{maximum } (\begin{cases} 0 & j+k>i \\ (i-j-k)+p & otherwise \end{cases}, \begin{cases} 0 & i<>j \\ p & otherwise \end{cases}, \begin{cases} 0 & i<>k \\ p & otherwise \end{cases},$$

$$\begin{cases} 0 & j=k \\ abs(j-k)+p & j \neq k \end{cases}, \begin{cases} 0 & i>0 \\ -i+p & otherwise \end{cases}, \begin{cases} 0 & j>0 \\ -j+p & otherwise \end{cases},$$

$$\begin{cases} 0 & k>0 \\ -k+p & otherwise \end{cases}, \text{minimum } (\begin{cases} 0 & tri<>1 \\ p & otherwise \end{cases}, \begin{cases} 0 & i+j<k \\ (i+j-k)+p & otherwise \end{cases}),$$

$$\text{minimum } (\begin{cases} 0 & tri<>2 \\ p & otherwise \end{cases}, \begin{cases} 0 & i+k<j \\ (j-i-k)+p & otherwise \end{cases}))$$

Trial #1: 827556323684491746 684491746 in Generation #157

Trial #2: 24076905 790891888 790891888 in Generation #50

The results of one complete run of TDGen on the triangle classification program are in table 5.7. The integers in the table represent the test case that achieves the branch coverage percentage progress. The population size for the GA is 100, which means 100 runs per generation..

Table 5.7 Test Cases that Traverse Untested Branches for Triangle Classification Program

Gen#	Statement #	Integer 1	Integer 2	Integer 3	Branch Coverage
1	4	-1003681762	-1789385610	1053597848	20.0%
1	14	270799300	1034857710	266378176	60.0%
1	15	1028865792	480254158	1012693452	63.3%
154	9, 26	461096133	1980897956	461096133	80.0%
338	7, 9, 11, 19	1070096700	1070096700	1070096700	83.3%
496	9, 23	400533888	416625066	400533888	86.7%

972	7, 21	1607086370	1607086370	331820571	93.3%
1136	11, 25	548879073	1915612299	1915612299	100.0%

Comparing to the result from Michael [2001] in Table 5.5, although more runs of the program are needed, TDGen got the full branch coverage of the triangle program.

6. Empirical Results

In order to determine the effectiveness of TDGen, a number of experiments on several programs are done and results are reported. The programs tested with TDGen are as follows:

1. Bubble sort
2. Greatest common divisor
3. *Conversion of a hexadecimal number to a decimal number*
4. *Bonus calculation*
5. *Quadratic formula*
6. *Triangle Classification*

For each programs, 10 runs of TDGen are performed, as well as 10 runs of random testing for the purpose of comparison. Although we use branch coverage in TDGen, results of both statement coverage and branch coverage are generated. Results of statement coverage of TDGen are only for the reference purpose, and we believe with some adjustment of TDGen, better performance of TDGen with statement coverage could be obtained.

The GA in TDGen uses 100 individuals per iteration/generation, and the maximum of the GA to give up satisfying a constraint and go to the next constraint is 300 generations. We choose 0.4 as the crossover probability and 0.1 as the mutation probability in the experiments. To produce a fair comparison, random testing is designed to randomly generate 100 test cases in an iteration, which is the size of the population of the GA.

The results of the experiments on the first two programs don't give much difference between TDGen and random testing, and both statement coverage and branch coverage are achieved within first several iterations. The reason is that the source code of the programs doesn't involve nested conditional structures, or the predicates of conditional structures and loops can be easily satisfied. The code of the programs is mainly straight lines or branches that are easily traversed by random test cases.

The results of the programs #3 to #6 in the list, hex-dec conversion, bonus calculation, quadratic formula and triangle classification, however, show the difference between TDGen and random testing. The source code of these four programs is provided in the Appendix. Table 6.1 summarizes the test results of the programs using TDGen and random testing. Mean, minimum and maximum number of generations to achieve the highest statement coverage and branch coverage are showed in the result table.

Table 6.1 Results of TDGen on the Four Programs

Program	Generator	Total Runs	Max. Statement / Branch Coverage	Mean	Min	Max
Hex-Dec	TDGen	10	96.43/92.85	15	8	25
	Random	10	96.43/92.85	33	14	58
Bonus	TDGen	10	100/100	33	23	46
	Random	10	100/100	261	108	447
Quadratic	TDGen	10	100/100	267	171	323
	random	10	76.67/77.78	1	1	1
Triangle	TDGen	10	100/100	973	644	1534
	Random	10	62.5/63.33	2	1	5

In the table, TDGen and random testing both reached the same highest statement and branch coverage for Hex-Dec Conversion program and Bonus program. However, random testing used an average of 33 iterations (100 cases / iteration), i.e. 33 generations in the GA process, to reach the maximum coverage for the Hex-Dec program, while it only took TDGen 15 iterations on average, and the minimum and maximum iteration that TDGen required to achieve the highest coverage are lower than those of random testing. For the bonus program, the difference is even bigger. TDGen only used approximately as one eighth of the iterations as random testing to reach 100% coverage, and the maximum iterations that TDGen needed is less than half of the minimum iterations that random testing needed. In the cases of quadratic formula program and triangle classification program, random testing didn't have any improvement since the first several iterations, and the statement and branch coverage achievement were poor. Random testing was just simply doing fruitless randomly test data generation, and eventually gave up. Comparing to random test data generator, TDGen achieves 100% statement coverage and branch coverage.

For all the programs, figures are given to graphically show the difference of the coverage improvement between TDGen and random testing. In each figure, the horizontal axis is the number of iterations run by test data generators, and the vertical axis gives the coverage percentage achieved. For random testing, only the mean values are showed in the graph, and for the statement and branch coverage graphs of TDGen, minimum, maximum and mean achieved coverage percentage values are displayed.

Figure 6.1, 6.2 and 6.3 summarize graphically the results of the experiments using random testing and TDGen on Hex-Dec program. Although the highest achieved statement coverage and branch coverage are the same for both random testing and TDGen, random testing needs almost twice the effort that TDGen needs to achieve the highest coverage.

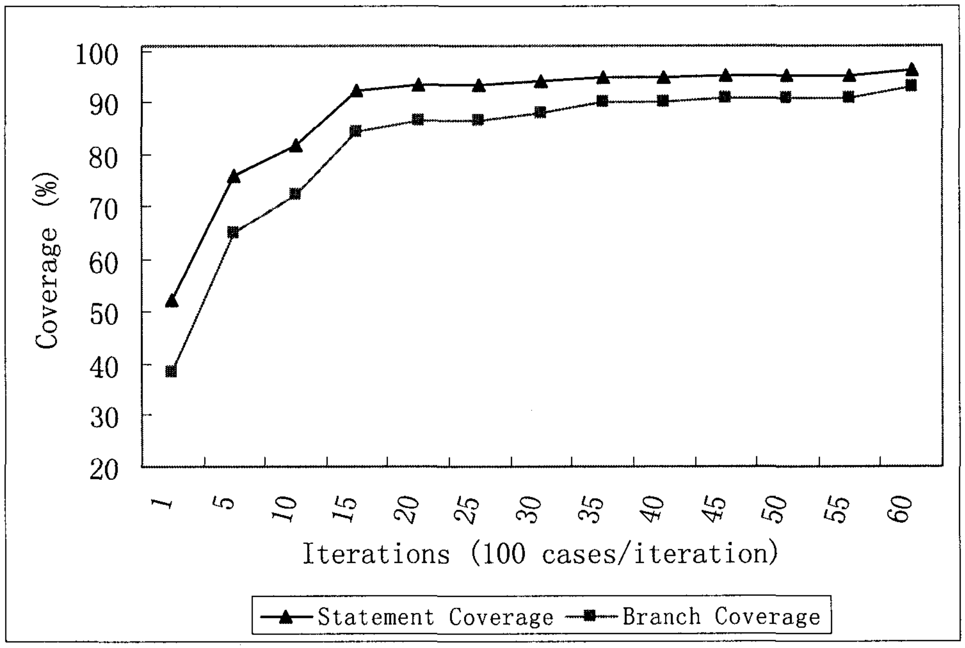

Figure 6.1 Statement Coverage and Branch Coverage of Random Testing on Hex-Dec Conversion Program

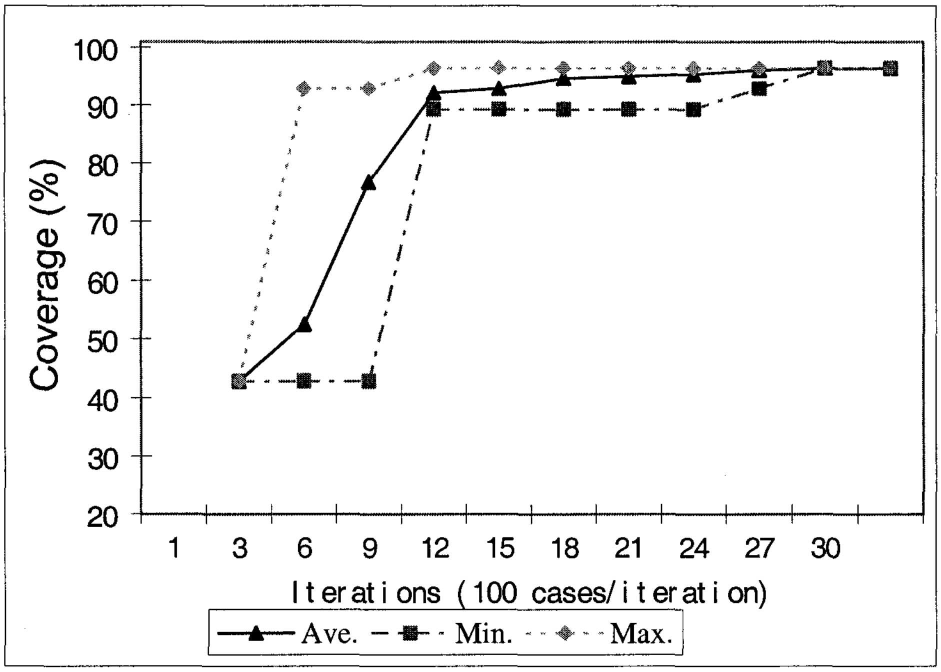

Figure 6.2 Statement Coverage of TDGen on Hex-Dec Conversion Program

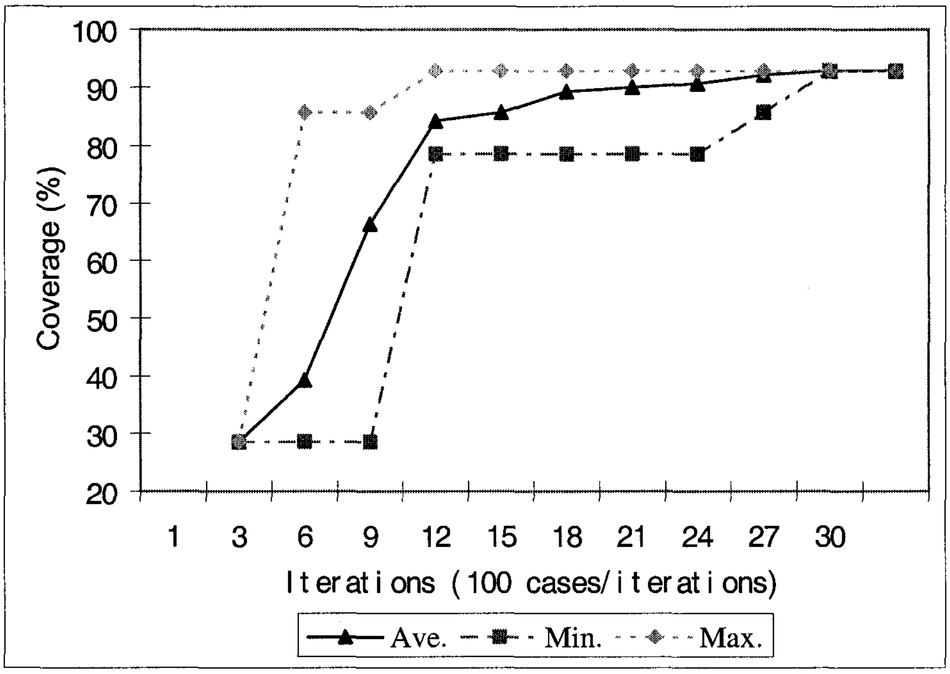

Figure 6.3 Branch Coverage of TDGen on Hex-Dec Conversion Program

Figure 6.4 to 6.6 summarize the results of experiments using random testing and TDGen on Bonus program.

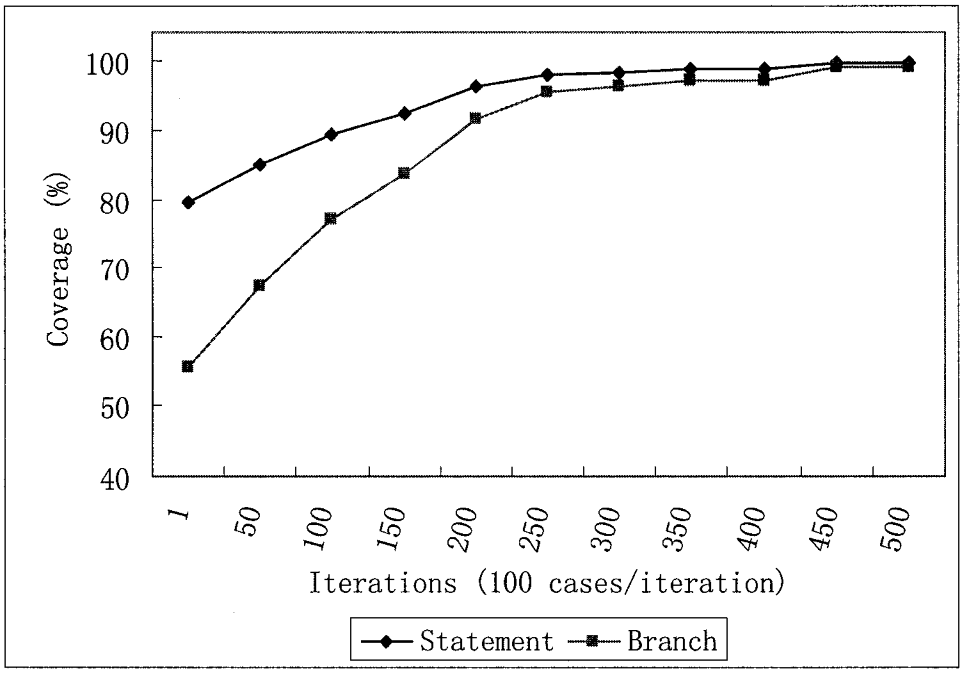

Figure 6.4 Statement Coverage and Branch Coverage of Random Testing on Bonus Program

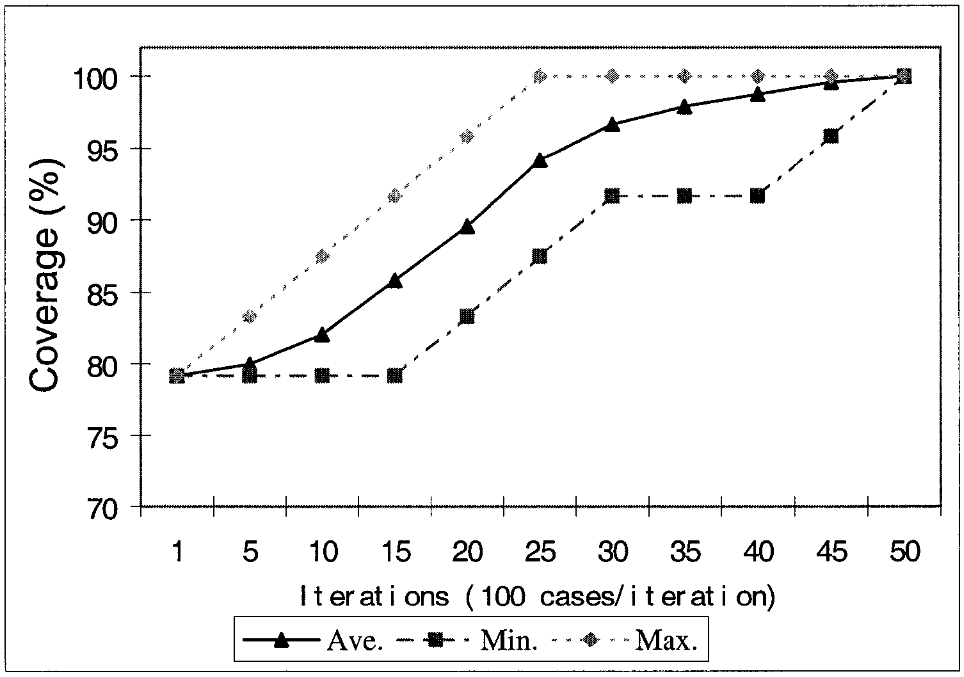

Figure 6.5 Statement Coverage of TDGen on Bonus Program

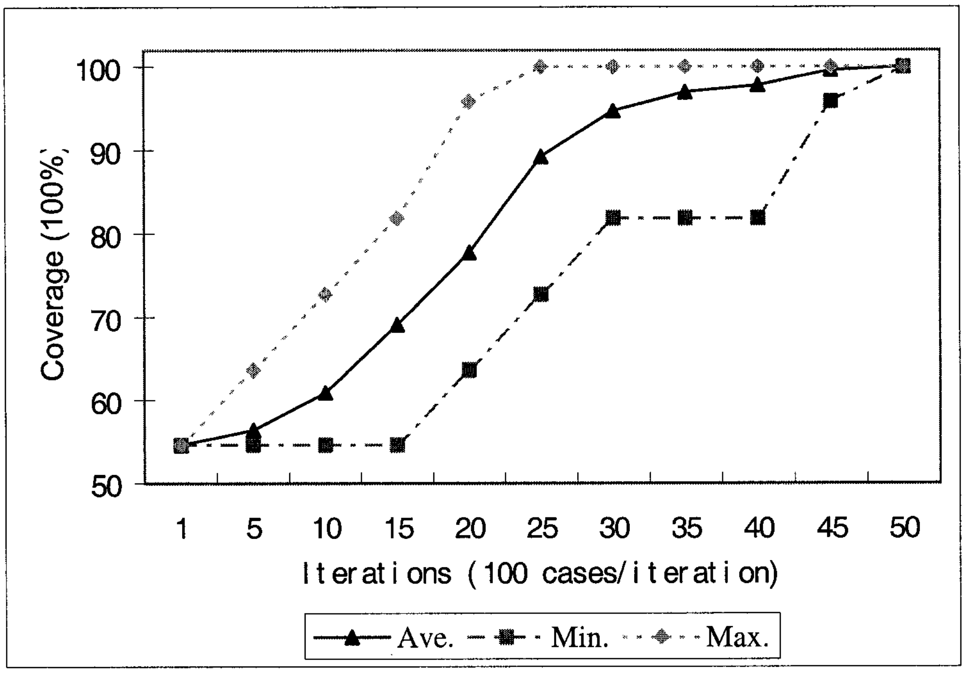

Figure 6.6 Branch Coverage of TDGen on Bonus Program

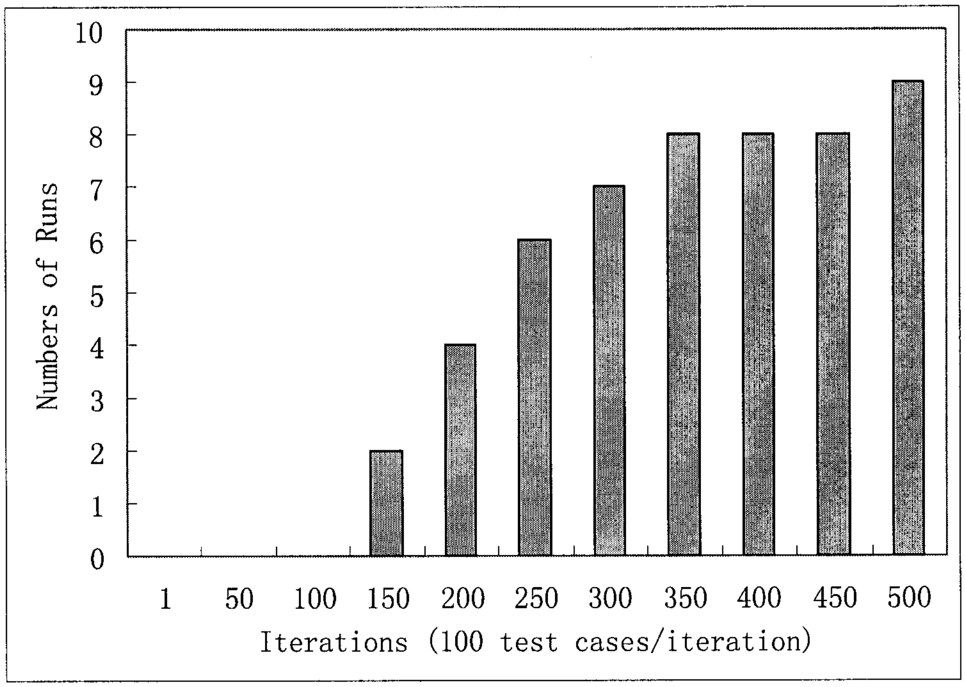

Figure 6.7 Cumulative Frequencies of Runs to Achieve 100% Statement and Branch Coverage of Random Testing on Bonus Program

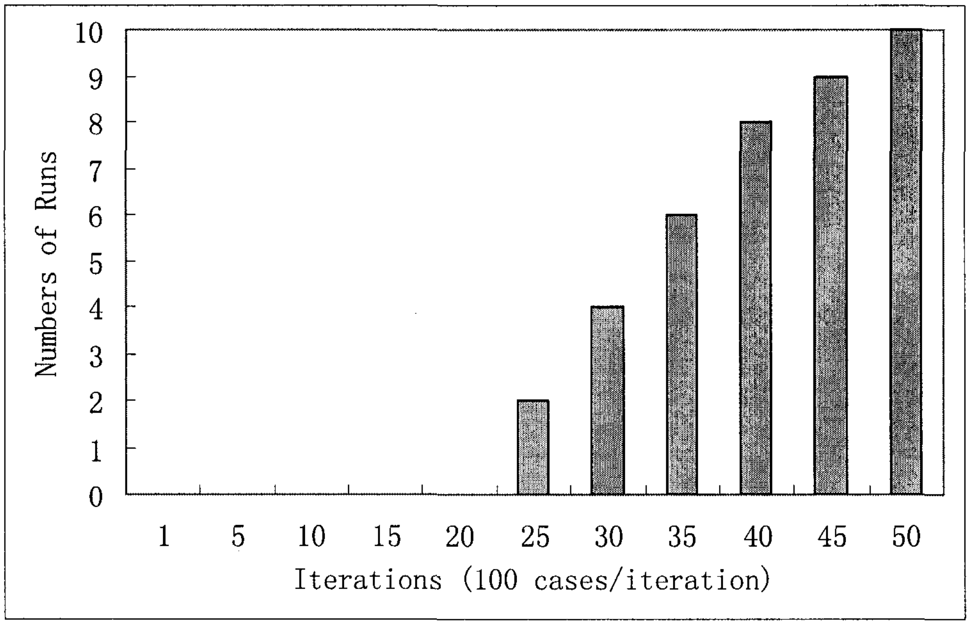

Figure 6.8 Cumulative Frequencies of Runs to Achieve 100% Statement and Branch Coverage of TDGen on Bonus Program

Figure 6.7 and 6.8 are the graphs showing cumulative frequencies of runs to achieve 100% coverage. The difference in the results of the bonus program is quite obvious. The range of runs achieving full coverage of TDGen is from 23 to 46. By contrast, the range for random testing is from 108 to 447, which is far greater than that of TDGen. The coverage graph and cumulative frequency figures of random testing show that there is one run failing to achieve full coverage within 500 iterations. The results of random testing indicate random testing has great difficulty working on large search space, and the chance of random testing to get the solution depends on the ratio of the domain of the solution to the total search space. The performance of random testing is quite unpredictable and good results depend on serendipity.

Results of random testing and TDGen on quadratic formula program are provided in Figure 6.9 to 6.11. Figure 6.9 shows that random testing failed to achieve any progress after the first iteration. In the program, inputs are real numbers, which have continuous search space. In this case, the chance of random testing to satisfy *a=0* is almost zero when *a* is a real number variable. On the contrary, with the assistance of GA and PDG analysis, TDGen achieved full statement and branch coverage within 350 iterations for all 10 runs.

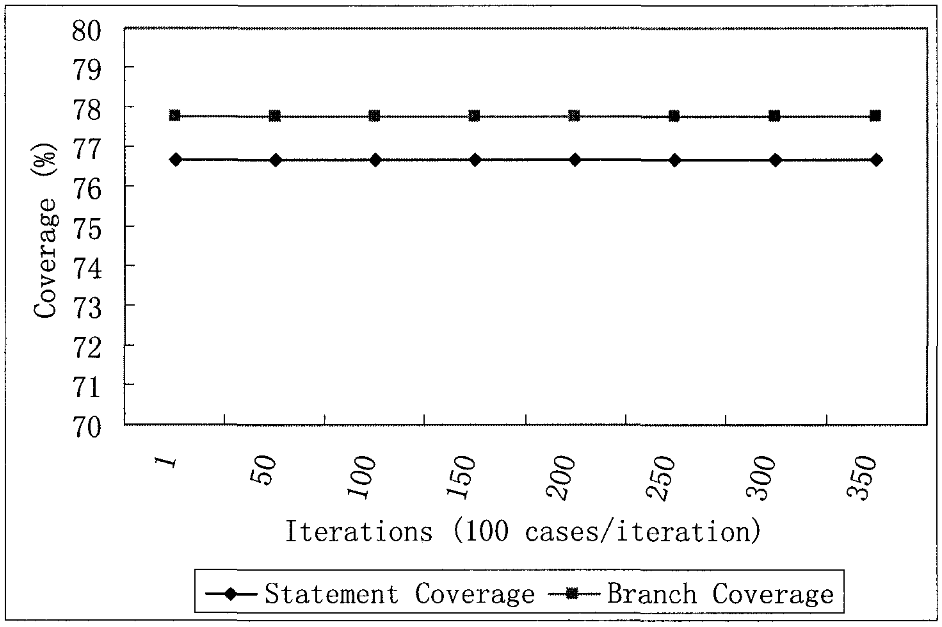

Figure 6.9 Statement Coverage and Branch Coverage of Random Testing on Quadratic Formula Program

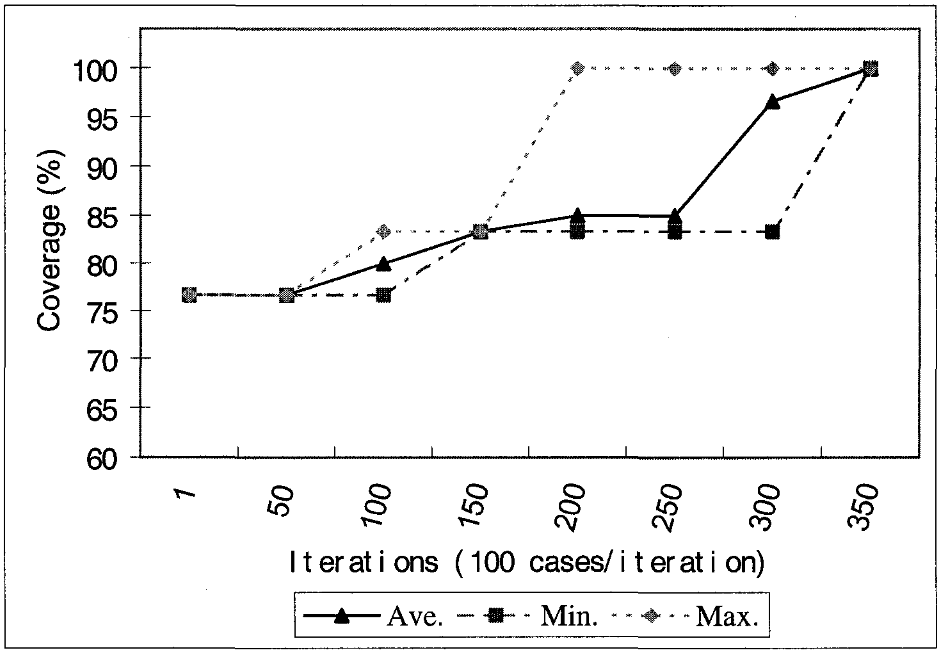

Figure 6.10 Statement Coverage of TDGen on Quadratic Formula Program

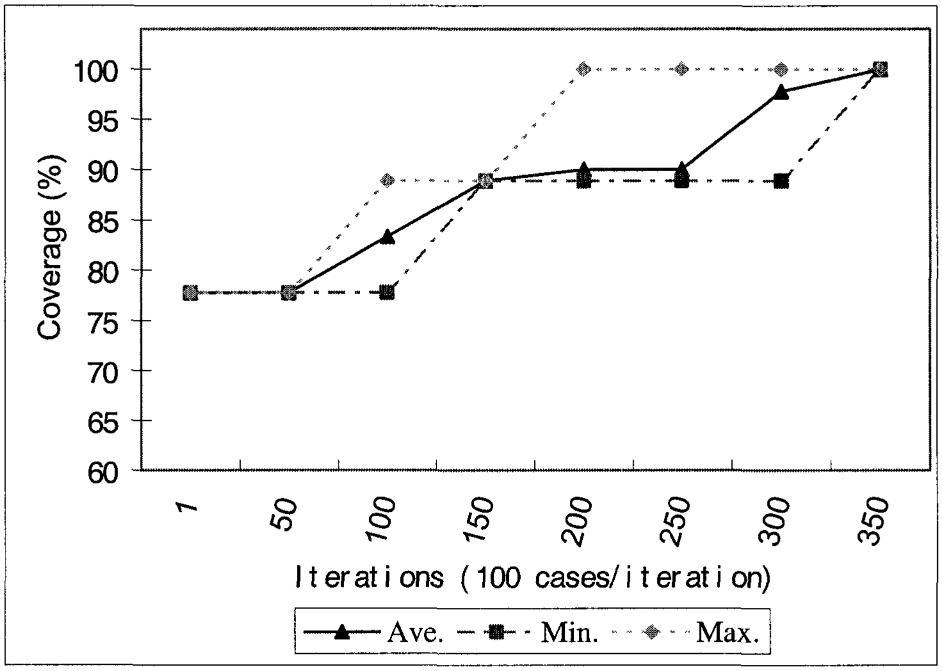

Figure 6.11 Branch Coverage of TDGen on Quadratic Formula Program

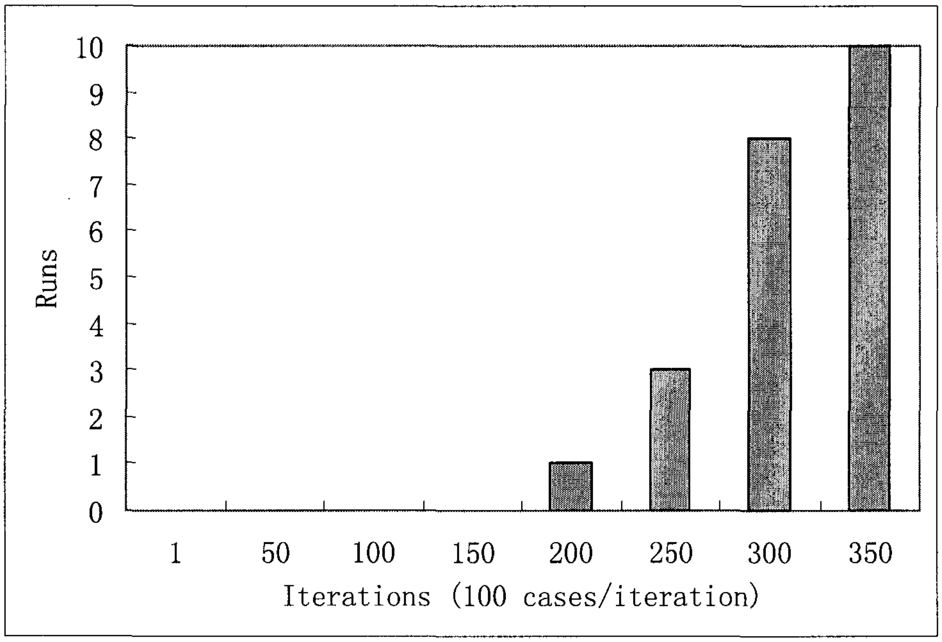

Figure 6.12 Cumulative Frequencies of Runs with 100% Statement and Branch Coverage of TDGen on Quadratic Formula Program

Figure 6.12 is the Cumulative Frequencies of Runs with 100% Statement and Branch Coverage of TDGen on Quadratic Formula Program. The best run got

100% coverage at 171st iteration, and it took the worst run 323rd iterations to get full coverage.

Figure 6.13 to 6. 15 are graphs of results using random testing and TDGen on triangle classification program. Random testing graph shows random testing was wasting effort during most of the time after it got the initial coverage percentage. Because of nested conditional statements and the variable with enumerated type in predicates in the triangle program, the chance that random test data generator to get lucky is very slim. The result graphs of TDGen indicate that TDGen kept getting gradual progress during all the process, and the difference between the worse case and best case is fairly reasonable.

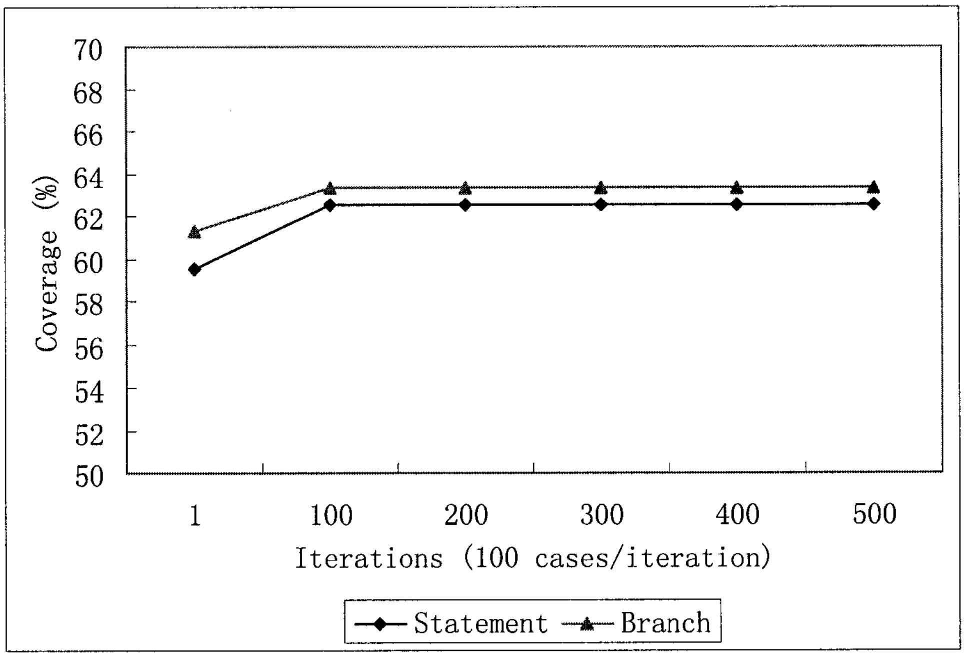

Figure 6.13 Statement Coverage and Branch Coverage of Random Testing on Triangle Classification Program

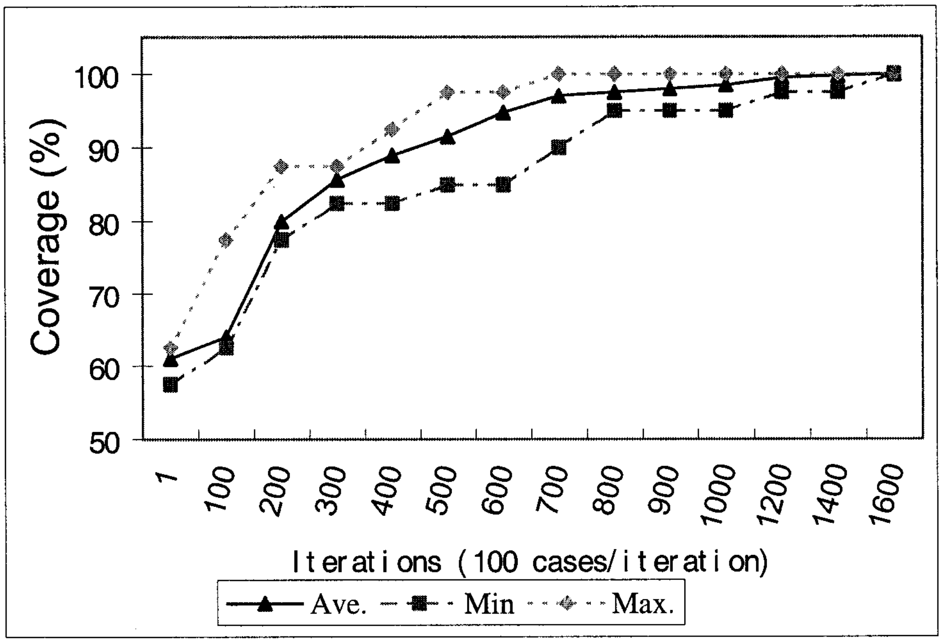

Figure 6.14 Statement Coverage of TDGen on Triangle Classification Program

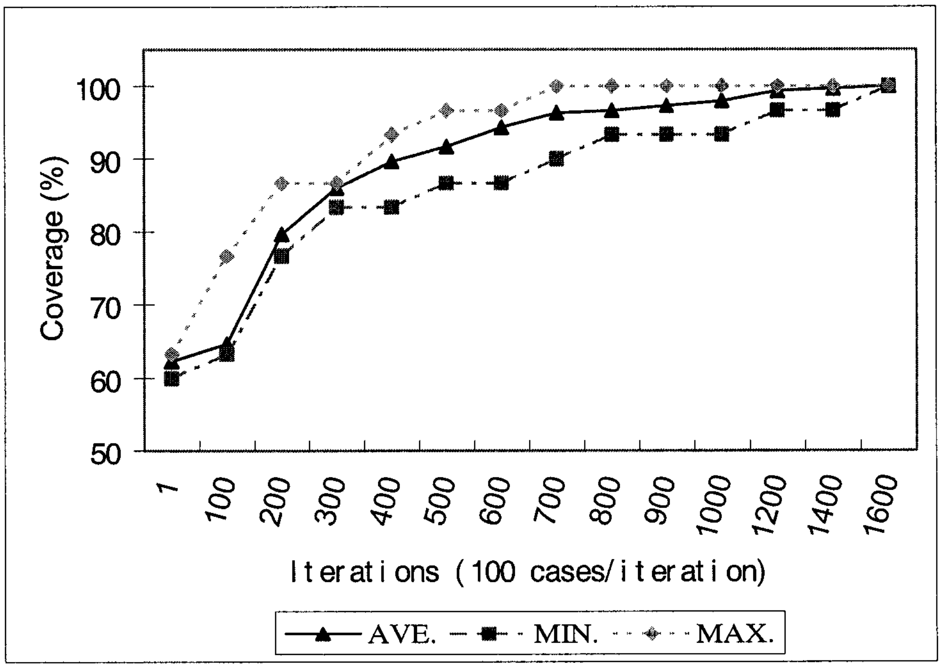

Figure 6.15 Branch Coverage of TDGen on Triangle Classification Program

Table 6.2 Statement Coverage Results of TDGen on Triangle Classification Program

Statement Coverage (%)		Iterations													
		1	100	200	300	400	500	600	700	800	900	1000	1200	1400	1600
runs	1	62.5	62.5	80	87.5	90	90	95	95	95	95	95	97.5	100	100
	2	57.5	62.5	80	87.5	90	90	97.5	100	100	100	100	100	100	100
	3	62.5	62.5	80	82.5	85	95	95	95	95	97.5	100	100	100	100
	4	62.5	62.5	77.5	85	92.5	97.5	97.5	97.5	97.5	97.5	97.5	100	100	100
	5	62.5	77.5	77.5	85	85	92.5	92.5	97.5	97.5	97.5	100	100	100	100
	6	62.5	62.5	77.5	85	92.5	92.5	97.5	100	100	100	100	100	100	100
	7	57.5	62.5	80	87.5	90	90	97.5	100	100	100	100	100	100	100
	8	57.5	62.5	80	87.5	90	90	92.5	97.5	97.5	97.5	97.5	100	100	100
	9	62.5	62.5	80	82.5	82.5	85	85	90	95	95	95	97.5	97.5	100
	10	62.5	62.5	87.5	87.5	92.5	92.5	97.5	97.5	97.5	100	100	100	100	100
Ave.		61	64	80	85.8	89	91.5	94.8	97	97.5	98	98.5	99.5	99.8	100
Min.		57.5	62.5	77.5	82.5	82.5	85	85	90	95	95	95	97.5	97.5	100
Max.		62.5	77.5	87.5	87.5	92.5	97.5	97.5	100	100	100	100	100	100	100

Table 6.2 and 6.3 give all coverage results of the 0 runs of TDGen on triangle classification program. These two tables also indicate that TDGen was getting progress gradually on statement and branch coverage.

Table 6.3 Branch Coverage of TDGen on Triangle Classification Program

Branch Coverage (%)		Iterations													
		1	100	200	300	400	500	600	700	800	900	1000	1200	1400	1600
runs	1	63.3	63.3	80	86.7	90	90	93.3	93.3	93.3	93.3	93.3	96.7	100	100
	2	60	63.3	80	86.7	90	90	96.7	100	100	100	100	100	100	100
	3	63.3	63.3	80	83.3	86.7	93.3	93.3	93.3	93.3	96.7	100	100	100	100
	4	63.3	63.3	76.7	86.7	93.3	96.7	96.7	96.7	96.7	96.7	96.7	100	100	100
	5	63.3	76.7	76.7	86.7	86.7	93.3	93.3	96.7	96.7	96.7	100	100	100	100
	6	63.3	63.3	76.7	86.7	93.3	93.3	96.7	100	100	100	100	100	100	100
	7	60	63.3	80	86.7	90	90	96.7	100	100	100	100	100	100	100
	8	60	63.3	80	86.7	90	90	93.3	96.7	96.7	96.7	96.7	100	100	100
	9	63.3	63.3	80	83.3	83.3	86.7	86.7	90	93.3	93.3	93.3	96.7	96.7	100
	10	63.3	63.3	86.7	86.7	93.3	93.3	96.7	96.7	96.7	100	100	100	100	100
AVE.		62.3	64.7	79.7	86	89.7	91.7	94.3	96.3	96.7	97.3	98	99.3	99.7	100
MIN.		60	63.3	76.7	83.3	83.3	86.7	86.7	90	93.3	93.3	93.3	96.7	96.7	100
MAX.		63.3	76.7	86.7	86.7	93.3	96.7	96.7	100	100	100	100	100	100	100

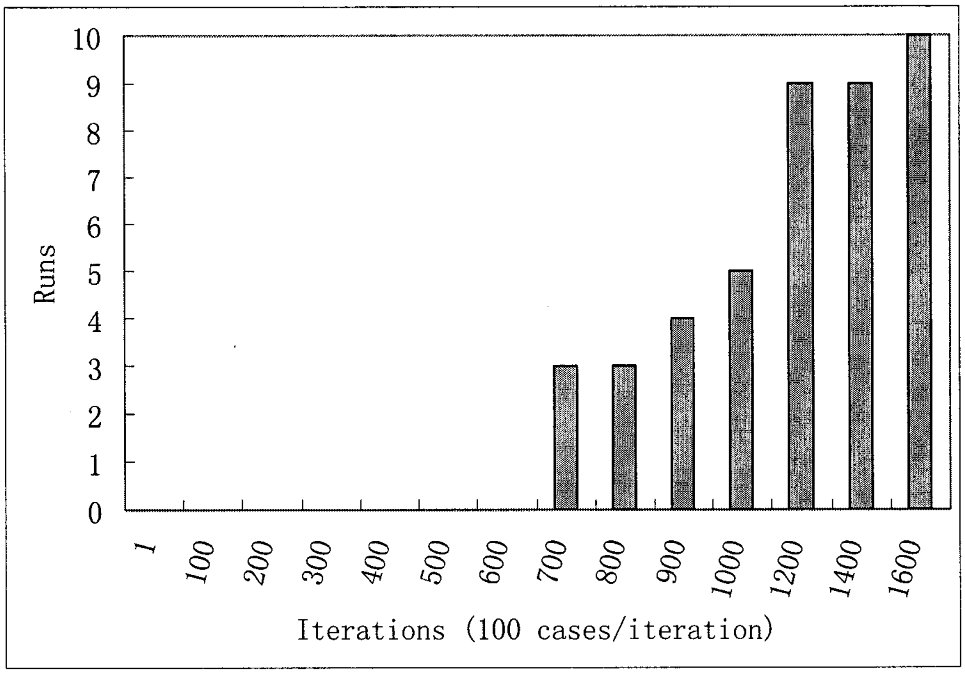

Figure 6.16 Cumulative Frequencies of Runs with 100% Statement and Branch Coverage of TDGen on Triangle Classification Program

Figure 6.16 shows cumulative frequencies of runs with 100% statement and branch coverage. The range of number of iterations that TDGen took to achieve 100% coverage is approximately from 700 to 1600. The reasons of the difference may be: 1) when TDGen uses data dependence analysis to try different paths, there might be permutation operations in the constraint obtained. TDGen then randomly chooses a permutation, so the faster TDGen can get to the right path, the less iterations it needs; 2) genetic algorithm is a dynamic process with randomness in it, and the speed of the evolutional process to converge to the solution varies from one trial to another.

The experiments of above programs show that TDGen outperforms random testing on programs with some complexity. The performance of random testing varies from one program to another, and in some case, its success depends on coincidence. On the contrary, the performance of TDGen is solid, and it keeps achieving progress on statement coverage or branch coverage during the whole process. For those programs with little complexity, TDGen and random testing both achieve 100% statement and branch coverage immediately. However, for those programs with nested conditions and variables with enumerated type in predicates that random testing has difficulty to handle, TDGen has better performance with the help of program dependence graph and genetic algorithm. Experiments also show that random testing has trouble with large search space, and the GA of TDGen is much more efficient working on large search space than random testing.

Three programs that GADGET failed to get 100% condition-decision coverage are tested using TDGen with the change from branch coverage to condition-decision coverage as the adequacy criterion. Table 6.4 shows the coverage percentages that GA in GADGET and GA in TDGen achieved.

Table 6.4 Results on Three Programs Using Condition-decision Coverage

	GA in GADGET	GA in TDGen
Binary Search	70%	100%
Quadratic	75%	100%
Triangle	94.29%	100%

With the PDG analysis, TDGen achieved 100% coverage for all three programs. The GA in TDGen works more effectively than the GA in GADGET, and TDGen outperforms GADGET in all three cases.

7. Conclusion and Future Works

This thesis presents a new approach for automatic test data generation using a genetic algorithm directed by program dependence graphs of the tested program. Two metrics are used in the proposed approach to be the criterion for target branch selection, and we believe the introduction of the target selection criterion can be helpful when testing on large programs. Path selection algorithm with program dependence analysis and the use of the genetic algorithm are the key techniques for TDGen to be successful.

Generating test cases using program dependence graphs and GA is proved to outperform random testing in the experiments of testing several programs, and it achieves higher coverage than GADGET on testing the triangle program. As compared to the previous automatic test data generation researches, we believe that TDGen is more likely to find test cases that other test data generation methods may be not able to get.

Several areas of possible future research are presented below:

- Although only branch coverage is chosen as the test adequacy criteria, the new approach can also be extended to other test criteria, such as path coverage and condition-decision coverage.
- Not all crossover and mutation operators have been tested to tune the GA to achieve the optimization. We believe there is some room to tune the GA in TDGen to get better performance with less computational cost.
- TDGen is just a prototype of an automatic test data generator, and there is still some work to do to make it a complete automatic test data generation tool. When a complete tool is developed, it will enable us to investigate the performance of TDGen on large programs.

Test data generation is a very complex problem and finding a thorough and perfect solution is extremely difficult. Any technique for automatic test data generation has limitations. TDGen has great potential in the area of automatic test data generation. Its power lies in that it inherits the simplicity and flexibility of genetic algorithms, while providing relatively more static analysis information of the tested programs to the genetic algorithm to make it work effectively and efficiently.

Bibliography

Anderson [2001] Paul Anderson, Tim Teitelbaum. Software Inspection Using CodeSurfer. Proceeding of the First Workshop on Inspection in Software Engineering, 2001

Back [2000] T. Back, D. B. Fogel, Z. Michalewicz, (eds.), Evolutionary Computations I, Institute of Physics Publishing, Bristol, 2000.

Barker [1985] J. E. Barker. Adaptive selection methods for genetic algorithms. In Proceedings of the First International Conference on Genetic Algorithms and Their Applications. Morgan Kaufmann, 1985.

Clarke [1985] A. Clarke and D. J. Richardson. Applications of Symbolic Evaluation. The Journal of Systems and Software, Vol. 5, No. 1, pp. 15-35, Jan., 1985

Darringer [1978] J. A. Darringer and J. C. King. Applications of Symbolic Execution to Program Testing. IEEE computer, Vol. 11, No. 4, April 1978.

Davis [1991] L. Davis. Handbook of Genetic Algorithms. Van Nostrand Reinhold, New York, 1991.

De Jong [1975] K. A. De Jong. An Analysis of the Behavior of a Class of Genetic Adaptive Systems. PhD thesis, University of Michigan, 1975.

DeMillo [1991] Richard A. DeMillo, A. Jefferson Offutt. Constraint-Based Automatic Test Data Generation. IEEE Transactions on Software Engineering, 17(9): 900-910, September 1991

Duran [1981] J. W. Duran, and S. Ntafos. A report on random testing, Proceedings 5th Int. Conf. on Software Engineering held in San Diego C.A., pp. 179-83, March 1981

Eshelman [1989] L. J. Eshelman, A. Caruana, J. D. Schaffer. Biases in the Crossover Landscape. Proceedings of the Third International Conference on Genetic Algorithms, pp. 86-91, 1989

Eshelman [1993] L. J. Eshelman and J. D. Schaffer. Real-Coded Genetic Algorithms and Interval-Schemata. Foundation of Genetic Algorithms, Morgan Kaufmann, pp. 187—202, 1993.

Fogel [1995] D. B. Fogel, Evolutionary Computation, Toward a New Philosophy of Machine Intelligence, IEEE Press, Piscataway, 1995.

Ferguson [1997] R. Ferguson and B. Korel. The Chaining Approach for Software Test Data Generation. ACM Transaction Software Engineering Methodology, vol. 5, No. 1, pp. 63-86, Jan. 1996.

Ferrante [1987] Jeanne Ferrante. Program Dependence Graph and its use in optimization. ACM Transaction on Programming languages and Systems, Vol. 9, No. 3, July 1987, 319-349.

Gallagher [1997] M. J. Gallagher and V. L. Narasimhan. Adtest: A Test Data Generation Suite for Ada Software Systems. IEEE Transactions. Software Engineering, Vol. 23, No. 8, pp. 473-484, Aug. 1997.

GrammaTech [1999] GrammaTech, Inc. Codesurfer user guide and reference manual.

Goldberg [1989] D. E. Goldberg. Genetic Algorithms in Searching, Optimization, and Machine Learning. Addison-Wesley, 1989

Goldberg [1991] D. E. Goldberg. Real-coded Genetic Algorithms, Virtual Alphabets, and Blocking. Complex Systems 5, 153-171.

Goldberg and Deb [1991] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. Foundations of Genetic Algorithms, pp. 69-93, San Mateo, 1991. Morgan Kaufmann.

Godefroid [2002] Patrice Godefroid and Sarfraz Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. Proceedings of TACAS'2002 , Grenoble, April 2002.

Herrera [1994] F. Herrera, E. Herrera-Viedma, M. Lozano and J. L. Verdegay. Fuzzy Tools to improve Genetic Algorithms. Proceedings of Second European Congress on Intelligent Techniques and Soft computing, pp. 1532-1539, 1994.

Herrera [1996] F. Herrera, M. Lozano and J. L. Verdegay. Tackling Real-Coded Genetic Algorithms: Operators and Tools for Behavioral Analysis.

Horwitz [1990] S. Horwitz, T. Reps, and D. Binkley, Interprocedural slicing using dependence graph, ACM Transactions on Programming Languages and Systems 12, 1, 1990, pp. 26-60

Horwitz [2002], Susan Horwitz. Tool support for improving test coverage. In Proceedings of ESOP 2002: European Symposium on Programming, (Grenoble, France, April 8-12, 2002).

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

Howden [1975] W. E. Howden. Methodology for the Generation of Program Test Data. IEEE Transactions on Software Engineering, (SE-24), May 1975

Ince [1987] D. C. Ince The automatic generation of test data. The Computer Journal, Volume 30, No. 1, 1987, pp. 63 - 69.

Korel [1990] B. Korel. Automatic Software Test Data Generation. IEEE transaction on software engineering volume: 16 pp.: 870-879, Aug. 1990.

Korel [1996] B. Korel. Automatic Test Data Generation for Programs with Procedures. Proceedings of International Symposium of Software Testing and Analysis, pp. 209-215, 1996.

Kuck [1981] D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe. Dependence Graphs and Compiler Optimizations. Conference Record of the Eighth ACM symposium on Principles of Programming Languages, pp. 207-218

Michael [2001] Christoph C. Michael, Gary McGraw and Michael Schatz. Generating Software Test Data by Evolution. IEEE transactions on software engineering, Vol. 27, No. 12, December 2001.

Michalewicz [1992] Genetic Algorithms + Data Structures = Evolution Programs. Srpinger-Verlag, New York, 1992.

Mitchell [1996] M. Mitchell. An Introduction to Genetic Algorithms. MIT Press, 1996.

Mühlenbein [1993] H. Mühlenbein and D. Schlierkamp-Voosen. Predictive Models for the Breeder Genetic Algorithm I. Continuous parameter Optimization. Evolutionary Computation 1, pp. 25-49, 1993.

Offutt [1988] A. Jefferson Offutt. Automatic Software Testing (Ph. D). Georgia Institute of Technology, 1988.

Offutt [1999] A. Jefferson Offutt, Zhenyi Jin, Jie Pan. The Dynamic Domain Reduction Procedure for Test Data Generation. Software--Practice and Experience, 29(2):167--193, January 1999.

Pargas [1999] Roy P. Pargas, Mary Jean Harrold and Robert R. Peck. Test-Data Generation Using Genetic Algorithms. Journal of Software Testing, Verification and Reliability, 1999.

Radcliffe [1991] N. J. Radcliffe. Forma Analysis and Random Respectful Recombination. Proceedings of the Fourth International Conference on Genetic Algorithms, pp. 222-229, 1991.

Roper [1994] Marc Roper. Software Testing. McGraw-Hill, c1994

Schlierkamp-Voosen [1994] D. Schlierkamp-Voosen. Strategy adaptation by Competition. Proceedings of Second European Congress on Intelligent Techniques and Soft Computing, pp. 1270-1274, 1994.

Syswerda [1989] G. Syswerda. Uniform Crossover in Genetic Algorithm. Proceedings of the Third International Conference on Genetic Algorithms, pp. 2-9, 1989.

Sturgis [1985] H. Sturgis. An Effective Test Strategy. Technical report CSL-85-8, Xerox Parc, Nov. 1985

Tracey [2000] Nigel James Tracey. A Search-Based Automated Test-Data Generation Framework for Safety-Critical Software (Ph. D). University of York, Sept. 2000.

Voas [1991] J. M. Voas, L. Morell, and K. W. Miller. Predicting Where Faults Can Hide From Testing. IEEE Software, Vol. 8, No. 2, pp. 41-58, 1991

Voigt [1994] H. M. Voigt and T. Anheyer. Modal Mutations in Evolutionary Algorithms. Proceedings of the First IEEE Conference on Evolutionary Computation, pp. 88-92, 1994.

Wright [1990] A. Wright. Genetic Algorithms for Real Parameter Optimization. Foundations of Genetic Algorithms, First Workshop on the Foundations of Genetic Algorithms and Classifier Systems, Morgan Kaufmann, Los Altos, CA, pp. 205-218, 1990.

Wright [1991] A. Wright. Genetic Algorithms for Real Parameter Optimization. Foundations of Genetic Algorithms, Morgan Kaufmann Publishers, pp. 205-218, 1991.

Yang [1995] X. Yang., B. F. Jones, and D. Eyres. The automatic generation of software test data from Z specifications, Research Project Report III, CS-95-2, February 1995

Appendix

Program Listings:

This Appendix contains listings of C programs. They are used for experiments of which the results are presented in Chapter 6. They are Hex-Dec Conversion Program, Bonus program, Quadratic Formula Program and Triangle Classification Program. Some comments are given in some programs. The statements are numbered based on coverage results generated by *gcov* (a test coverage program) of GCC. The numbering of triangle program is different from that used in Chapter 5.

Hex-Dec Conversion Program

```
        # include <stdio.h>
        # define MAX 5
        int htoi (char[]);

        int main ()
1       {
2               char t[MAX];
3               char c;
4               int i;
5               i=0;
6               printf("Input a hex number:");
7               while ((c=getchar())!='\n')
                {
8                       if(c>='0' &&c<='9'||c>='a'&&c<='f'||c>='A'&&c<='F')
                        {
9                               t[i]=c;
10                              i++;
                        }
                        else
                        {
11                              printf("Not a valid hex number\n");
12                              return 0;
                        }
                }
13              if (i<=MAX)
                {
```

```
14                      t[i]='\n';
15                      printf("decimal number: %d\n",htoi(t));
                }
                else
16              printf("Maximum 7 digits of hex number\n");
17              return 0;
        }

        long int htoi(char s[])
18      {       int j;
19              long int n;
20              n=0;
21              for (j=0;s[j]!='\n';j++)
                {
22                      if (s[j]>='0'&&s[j]<='9')
23                              n=n*16+s[j]-'0';
24                      if (s[j]>='a'&&s[j]<='f')
25                              n=n*16+s[j]-'a'+10;
26                      if (s[j]>='A'&&s[j]<='F')
27                              n=n*16+s[j]-'A'+10;
                }
28              return (n);

                                        }
```

Bonus Program

```
# include <stdio.h>

int main()
{
    int i;
    double bonus, bon1, bon2, bon4, bon6, bon10;
    bon1=100000*0.1;              //bonus for 0 to 100000
    bon2=bon1+100000*0.075;  //bonus for 100000 to 200000
    bon4=bon2+200000*0.05;    //bonus for 200000 to 400000
    bon6=bon4+200000*0.03;    //bonus for 400000 to 600000
    bon10=bon6+400000*0.015; //bonus for 600000 to 1000000
    printf ("input the profit :");
    scanf ("%d", &i);
    if (i<=100000)
          bonus=i*0.1;
    else if (i<=200000)
          bonus=bon1+(i-100000)*0.075;
    else if (i<=400000)
          bonus=bon2+(i-200000)*0.05;
    else if (i<=600000)
          bonus=bon4+(i-400000)*0.03;
    else if (i<=1000000)
          bonus=bon6+(i-600000)*0.015;
    else if (i>1000000)
          bonus=bon10+(i-1000000)*0.01;
    printf ("Bonus for profit amount of %d is %10.2f. \n", i, bonus);
    return 0;

}
```

Quadratic Formula Program

```
# include <math.h>
# include <stdio.h>

double x1, x2, disc, p, q;

int greater_than_zero (float a, float b) {
        x1=(-b+sqrt (disc))/(2*a);
        x2=(-b-sqrt (disc))/(2*a);
        return 0;
}

int equal_to_zero (float a, float b) {
        x1=x2=(-b)/(2*a);
        return 0;
}

 int smaller_than_zero (float a, float b) {
        p=-b/(2*a);
        q=sqrt(-disc)/(2*a);
        return 0;
 }

  int main() {
        float a, b, c;
        printf ("\nInput a, b, c:\n");
        scanf ("%f%f%f", &a, &b, &c);
        if (a==0)
        {
                printf ("\nNot a quadratic equation.\n");
                return 0;
        }
        printf ("\nequation: %5.2f*x*x+%5.2f*x+%5.2f=0\n", a, b, c);
        disc=b*b-4*a*c;
        printf ("root:\n");
        if (disc>0)
        {
                greater_than_zero(a, b);
                printf ("x1=%5.2f \nx2=%5.2f\n", x1, x2);
        }
        else if (disc==0)
        {
                equal_to_zero (a, b);
                printf ("x1=%5.2f \nx2=%5.2f\n", x1, x2);
        }
        else
        {
                smaller_than_zero (a, b);
                printf ("x1=%5.2f+%5.2fi \nx2=%5.2f-%5.2fi\n", p, q, p, q);
        }
        return 0;
        }
```

Triangle Classification Program

```
        #include <stdio.h>

        int triang (int i, int j, int k)
1       {
2               int tri;
3               if ((i<=0)||(j<=0)||(k<=0))
4                       return (4);
5               tri=0;
6               if (i==j)
7                       tri+=1;
8               if (i==k)
9               tri+=2;
10              if (j==k)
11                      tri+=3;
12              if (tri==0)
                {
13                      if ((i+j<=k)||(j+k<=i)||(i+k<=j))
14                              tri=4;
                        else
15                              tri=1;
16                      return tri;
                }
17              if (tri>3)
18                      tri=3;
19              else if ((tri==1)&&(i+j>k))
20                      tri=2;
21              else if ((tri==2)&&(i+k>j))
22                      tri=2;
23              else if ((tri==3)&&(j+k>i))
24                      tri=2;
                        else
25                      tri=4;
26              return tri;
        }

        int main (void)
27      {
28              int a, b, c, t;
29              printf("");
30              scanf ("%d %d %d", &a,&b,&c);
31              t=triang (a,b,c);
32              if (t==1)
33                      printf("scalene\n");
```

```
    if (t==2)
        printf("isosceles\n");
    if (t==3)
        printf("equilateral\n");
    if (t==4)
        printf("not triangle\n");
    return (0);
}
```