# AUTOMATIC TEST DATA GENERATION USING GENETIC ALGORITHM AND PROGRAM DEPENDENCE GRAPH

**Ms.Roshni Rajkumari**
KSR College of Technology,
Anna University,
Tiruchengode-637 215.
E-mail: rajkumari.roshni@gmail.com

**Dr.B.G.Geetha**
Professor & Head, Dept. of CSE.
KSR College of Technology,
Tiruchengode-637 215.
E-mail:geethaksrct@.gmail.com

**Abstract**

Software testing is probably the most complex task in the software development cycle. It is one of the most time-consuming and frustrating process. The complexity of software systems has been increasing dramatically in the past decade, and software testing as a labour-intensive component is becoming more and more expensive. With the complexity of the software, the cost of testing software is also increased. Thus with automatic test data generation the cost of testing will dramatically be reduced. This paper uses a program dependence analysis and genetic algorithms to generate test data automatically.

***Keywords:*** Automatic Test Data Generation, Software Testing, Genetic Algorithm, Program Dependence Graph.

## 1. Introduction

Software testing is an expensive component of software development and maintenance. A particularly labor-intensive component of this process is the generation of test data to satisfy testing requirements. Automation of testing is a crucial concern. Through automation, large-scale thorough testing can become practical and scalable.However, the automated generation of test cases presents challenges. With the complexity of the software, the cost consumed by verification grows with the increase of the defect rate. Currently, over half of all errors are not found until 'down-stream' in the development process. This process deteoriates as we move into an e-business environment business environment is characterized by incredibility short period called the "Internet Time".

This pace is driven by market conditions and the ever-evolving technology utilized in these types of systems. Given the gap between, the current level of success obtained by V&V technology and the required success, at a revolutionary change rather than incremental change is required to address the issue. The only solution is to automate or at least partially automate many V & V activities. This paper address the automatic generation of test cases from the source code.

## 2. Related Work

A small number of test-data techniques have already been automated: random, structural or path-oriented, goal oriented, analysis-oriented test-data generators. However, the limitations of these approaches have stopped their general acceptance. Random generators [4,13] create large amounts of test data; however, because no information exists about the testing objectives, the generators often fail to find data that satisfy the stated objectives of the testing process. A structural or path-oriented generator [7,14] first identifies a path for which test data is to be generated; unfortunately, the path is often infeasible causing the generator to fail to find an input that will traverse the path. Analysis-oriented generators [Korel 1996] have the ability to generate high quality test-data, but rely upon their designer having a great insight upon the domain of operation, and hence are not readily generalisable to arbitrary software systems. Hence, the project plans to explore goal-oriented generators to provide an industrial strength solution.

Although a number of different goal-oriented approaches and algorithms exist, it is difficult to judge exactly which approach represents the current state of the art. As with most branches of Software Engineering, empirical evaluation and comparison is extremely difficult and demanding and no easy answer exists. Hence, GADGET [11] is selected as a representative of the current state of the art in goal-oriented approaches, not because it was clearly superior, but because it had been reported in detail and hence provided a reasonable basis for allowing some limited comparison of relative performance. Hence, the following section provides a detailed overview of GADGET and its performance.

### 2.1 GADGET

GADGET[11], applies genetic algorithms to test data generation problem. GADGET attempts to satisfy the condition-decision coverage criterion. A condition is a statement that has a boolean value, and a decision is an expression that can change the flow of control of a program, such as an if or case statement. Condition-decision coverage requires that every branch in the program be taken, and that every condition in the code be tested as true at least once and as false 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 (Table 1). After this, the algorithm uses the coverage table to select a series of test requirements in sequence.

**Table 1.Example of coverage table**

| **Condition/decision** | **Status** | |
|---|---|---|
| | **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 the 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. When the GA works on a certain requirement, many other requirements are often coincidentally satisfied. For each of not completely satisfied test requirement, a fitness function $\Im$ is generated to map the problem to a minimization problem. Table 2 shows how $\Im$ is calculated for some typical relational operators in GADGET.

The advantage of GADGET is that it uses a GA insteadof gradient descent used by earlier approaches to avoid being stuck at the local minima. However, there are some problems; GADGET depends heavily on the serendipity of whether a specific location (line of code) 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 have not been reached are left behind until they are reached by coincidental satisfaction of some input generated by the GA trying to satisfy other conditions.

**Table 2.Operators for fitness functions**

| **Decision Type** | **Example** | **Fitness function** |
|---|---|---|
| Inequality | If(c>=d) … | $\Im(x) = \{d - c, if d \geq c;$ $0 otherwise$ |
| Equality | If(c=d) … | $\Im(x) = \| d - c \|$ |
| True/false value | If(c)… | $\Im(x) = \{1000, if c = false$ $0 otherwis$ |

**3. Test Data Generation using PDGs and GA**

A new approach, called TDGen, is been proposed which automatically generates test data using program dependence graphs and GAs to test programs based on branch coverage criterion. The idea of the approach is that path selection is crucial for testing many branches; therefore, static analysis is used to select 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 for 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 generators using only GA, the test data generator using both GA and PDG analysis gives the GA more constraints to work with during test data generation, so it is more effective and efficient.

Figure. 1 shows the system architecture and dataflow of TDGen. The coverage table  is established to record the branch information of the program under test, and keeps track of whether a branch is tested or not. When the coverage table is initialized with the seed inputs, the test data generator gets the next 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. 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 a new target will be selected from the coverage table for the next cycle of test data generation.

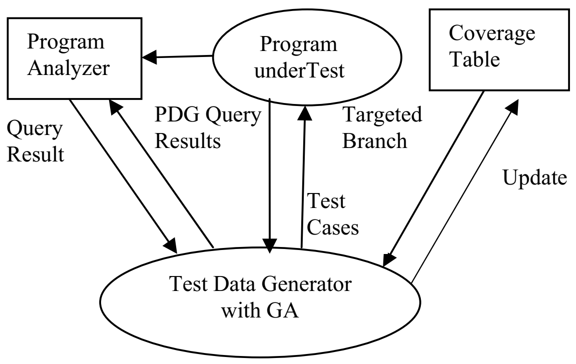

**Figure 1.System Architecture and Dataflow Diagram**

Figure.2 shows the algorithm of TDGen. A candidate solution is a set of test data, i.e. an array or a list of inputs for the program under test.

### 3.1 Coverage Table and Priority Ranking of Branches

A coverage table is established, showing the predicate number, program line numbers of predicates, predicate, true/false branch and branch coverage status. Using TDGen with branch coverage as the test adequacy criterion 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 Before starting to generate test data for the tested program, a seed input, generated randomly, is used to execute the program under test for the first time. Generally, running the program with the seed input will result in some branches being 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.

After the initial coverage table has been established, the next target branch is to be found out. Since traversing of different branches will have different contributions to the satisfaction of the selected criterion, the weight of each branch towards the branch coverage criterion should be seen. Higher priority is given to those branches that their traversing causes additional branches to be traversed or makes other branches easier to be traversed.

```
Procedure TDGen
Input:
        Program: a program to be tested
Output:
        covTable: the table to record information
and testing status
        testCases: set of test cases that are soln to
test corresponding  branches.
Variable declaration:
        pdgConstraint
        testRequirements
        curpopulation
        nextPopulation
        individual
        target
        maxIteration
        iterations
        iterationCounter
Begin
Create covTable;
Initialize covtable with results of random test data
generation;
Create testRequirements;
For(each entry of testRequirements)
{
  Target=current entry of testRequirement;
  Proceed PDG analysis for the target and convert
  the PDG analysis result to pdgConstraint;
  Initialise 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 the next
      population;
      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 2.Algorithm for TDGen**

#### 3.1.1 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.

### 3.1.2 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. 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.

## 3.2 Genetic Algorithm

### 3.2.1. Concept of evolutionary based algorithm

Genetic algorithm (GA) [6], is one of the most popular evolutionary-based algorithms. It has been successfully applied to numerous problems both at the level of structural and parametric optimization, and to software testing. GA is a search method utilizing the principles of natural selection and genetics. Concisely, GA operates on a set of candidate solutions, called a population, to a given problem. The candidate solutions are evaluated based on their ability to solve the problem. The results of the evaluation are used in a process of forming a new set of solutions. The choice of individuals that are passed to the next population is performed in a process called selection. This process is based on 'goodness' of candidate solutions. Additionally, genetic operators, i.e. crossover and mutation, are employed to modify selected candidates. Such sequence of actions is repeated until some final criterion is fulfilled.

### 3.2.2 Evolutionary-based test data generator

The test data generator initializes the first population for the GA. Each element of the population, which is a test case, is evaluated according to the fitness function, and given a fitness value.

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 utilize crossover operations to generate a new pair of offspring in the next generation. A crossover and a mutation probability are set up so that each pair of parents has a preset probability of doing the crossover operation, and each of the offspring after the crossover operations only mutates with the preset probability. If crossover is not utilized, both of the parents will be kept intact, and passed on to the mutation procedure. Single crossover is used for crossover operator. If a mutation is required upon a single individual, one of the three mutation operators: uniform random mutation, nonuniform random mutation and Mu'hlenbein's mutation, is randomly chosen as the mutation operator applied on the individual. The reason of randomly selecting a 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 would not degenerate from one generation to the next generation, and the best individual would not 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 repeats until the fitness function reaches 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 number of generations, or the values of fitness function stops getting smaller in a predetermined number 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. 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. 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.

## 4. Conclusion

A new approach for automatic test data generation using a genetic algorithm directed by program dependence graphs from the software under test is presented. It demonstrates the limitations of optimization only approaches, and demonstrates the advantage of a hybrid approach, where the program analysis techniques can compensate for some of the shortcomings of the goal-oriented strategy. Two metrics are used in the proposed approach to be the criterion for target branch selection, and the introduction of the target selection criterion can be helpful when testing on larger programs. Path selection algorithm with program dependence analysis and the use of the genetic algorithm are the key techniques within our system. Generating test cases using program dependence graphs and GAs is shown to outperform random testing; in addition, it is believed that TDGEN performs reasonably against other approaches (such as GADGET).

TDGen still possesses limited capabilities when compared to the requirements of an industrial strength automatic test generation engine. The system can be enhanced with the following features.

- Although only branch-type coverage measures are chosen as the test adequacy criteria, the new approach can also be extended to other test criteria, such as path 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 about the software under test to the genetic algorithm allowing it to work more effectively and efficiently.

## 5. References

1. Baresel, H. Pohlheim, S. Sadeghipour(2003), “Structural and Functional Sequence Test of Dynamic and State-Based Software with Evolutionary Algorithms”, GECCO, , pp. 2428–2441.
2. Jones, H. Sthamer, D. Eyres(1995), “The automatic generation of software test data sets using adaptive search techniques”, Proceedings of Third International Conference on Software Quality Management, Vol. 2, pp. 435–444.
3. Korel(1996), “Automatic Test Data Generation for programs with Procedures”, Proceedings of International Symposium of Software Testing and Analysis, pp. 209–215.
4. C.C. Michael, G. McGraw, M. Schatz(2001), “Generating Software Test Data by Evolution”, IEEE transactions on software engineering 27 (12) 1085–1110.
5. C.V. Ramamoorthy, S.F. Ho, W.T. Chen(1976), “On the generation of program test data”, IEEE Transactions on Software Engineering 2 (4) 293–300.
6. D.E. Goldberg1989, “Genetic Algorithms in Search, optimization and Machine Learning”, Addison-Wesley, Reading, MA.
7. L.A. Clarke(1976), “A system to generate test data and symbolically execute programs”, IEEE Transactions on Software Engineering 2 (3) 215–222.
8. N. Tracy, J. Clark, K. Mander(1998), “Automated program flaw finding using simulated annealing”, Proceedings of the 1998 International Symposium on Software Testing and Analysis.
9. N. Tracy, J. Clark, K. Mander, J. McDermid(2000), “Automated test-data generation for exception conditions”, Software Practice and Experience 30 (1) 61–79.
10. P. McMinn(2004), “Search-based software test data generation: a survey, Software Testing”, Verification and Reliability 14 (2) 105–156.
11. P. McMinn, M.Holcombe(2003), “The State Problem for Evolutionary Testing”, GECCO, pp. 2488–2498.
12. S. Horwitz2002, “Tool support for improving test coverage”. In Proceedings of ESOP 2002: European Symposium on Programming.
13. P. Thevenhod-Fosse, H. Waeselynck1998, “STATEMATE: applied to statistical software testing”, Proceedings of the 1998 International Symposium on Software Testing and Analysis,.
14. W.E. Howden(1977), “Symbolic testing and the DISSECT symbolic evaluation system”, IEEE Transactions on Software Engineering 3 (4) 266–278.

## Biography

**Miss. Roshni Rajkumari** is a student of K.S.Rangasamy College of Technology,Tiruchengode. She received her B.E.(Information Technology) from North Maharastra University in 2008 and her Masters Degree, M.E(Computer Science And Engineering) from K.S Rangasamy College of Technology. Her area of interest includes software engineering and computer networks.

**Dr.B.G.Geetha** has received B.E. Degree from Periyar Maniammai College of Technology for Women in Computer Science and Engineering in 1992, M.E. degree in Bharathair university in 2000 and PhD degree from Anna University in 2009. Currently she is working as a Professor & Head of Computer Science and Engineering Dept, K.S Rangasamy College of Technology, Tiruchengode and has 17 years of teaching experience. Her area of specialization includes software engineering and operating system.