Using Genetic Algorithm for Unit Testing Of Object Oriented Software

Nirmal Kumar Gupta
CSIS Group
Birla Institute of Technology & Science, Pilani
Pilani, India
nirmalgupta@bits-pilani.ac.in

Dr. Mukesh Kumar Rohil
CSIS Group
Birla Institute of Technology & Science, Pilani
Pilani, India
rohil@bits-pilani.ac.in

***Abstract*— Genetic algorithms have been successfully applied in the area of software testing. The demand for automation of test case generation in object oriented software testing is increasing. Genetic algorithms are well applied in procedural software testing but a little has been done in testing of object oriented software. In this paper, we propose a method to generate test cases for classes in object oriented software using a genetic programming approach. This method uses tree representation of statements in test cases. Strategies for encoding the test cases and using the objective function to evolve them as suitable test case are proposed.**

Keywords- Genetic Algorithm,; Object-Oriented Testing; Unit Testing; Chromosome; Mutation; Objective Function

I. INTRODUCTION

Testing is one of the vital activities which must be performed during software development. It is conducted by executing the program developed with test inputs and comparing the observed output with the expected one. Because the input space of the Software Under Test (SUT) might be very large, testing has to be conducted with a representative subset of test cases. To create relevant subset of test cases during software testing is the most critical activity. The test cases which are used to examine the SUT must posses an ability to expose the faults as well as test cases must be a representative subset of possible inputs. The quality and the significance of the overall test are directly affected by the set of test cases that are used during testing. Test data is used to create the test cases.

Generally test data is generated through a test data generation tool, while a test adequacy criterion assures the quality of test cases generated and gives information about the end of testing process. Many test data generators have been developed [1,2,3] each one using different kinds or variations of existing testing techniques.

The identification of good test cases generally follows some predefined testing criteria, for example code coverage [4], path coverage [1,5] statement coverage [2] and branch coverage [3]. In recent years some more advanced heuristic search techniques have been applied to software testing. These techniques are based on evolutionary algorithms, and their performance in finding test cases was found to be at least as good as random testing, but in many cases it is much better [6]. Commonly these testing techniques are referred as evolutionary testing [8].

Evolutionary Testing uses a kind of meta-heuristic search technique, the Genetic Algorithm (GA), to convert the task of test case generation into an optimal problem. Evolutionary Testing is used to search for optimal test parameter combinations that satisfy a predefined test criterion. This test criterion is represented by using a "cost function" that measures how well each of the automatically generated optimization parameters are satisfying the given test criterion.

This paper proposes a new approach to automate the generation of object-oriented unit test cases. Genetic programming is used to formulate the task of sequence generation as a search problem. This approach enables test case generation of object oriented classes by appropriately designing the objective function that is used to guide the genetic programming search. We will concentrate on the automated generation of test cases. One efficient way to achieve this is to use a random test generator. Random testing can be used to create a lot of test cases, but does not follow specifically any test coverage criteria.

This paper proposes a method to generate test cases for classes in object oriented software. In section II we introduce what are genetic algorithms in general while section III describes it with reference to a class testing. Section IV explains chromosomes and section V describes to encode them. Section VI, VII and VIII describe about selecting initial population, their crossover & mutation and then decoding them back to generate test cases respectively. Section IX describes about the objective function to guide the selection for next generation. Section X describes the experimental results and finally section XI describes limitations and future research directions.

II. GENETIC ALGORITHM FOR TESTING

Genetic Algorithms follow the concept of solution evolution by stochastically developing generations of solution populations using some given fitness function. They are particularly applicable to large, non-linear and possibly discrete in nature kind of problems. Evolutionary algorithms (EA) when applied for testing procedural software can be used to specifically look for test scenarios that cover certain branches of a program. These kinds of algorithms are based

on reproduction, evaluation and selection. Consider the flow chart in figure 1 which represents a standard GA that can be used for testing.

The GA in general has mainly four stages, which are evaluation, selection, crossover and mutation. The evaluation procedure measures the fitness of each individual solution (also called chromosome) in the population and assigns it a relative value based on the defining optimization (or search) criteria. The selection procedure randomly selects individuals of the current population for development of the next generation. Various alternative methods exist but all follow the idea that the fittest have a greater chance of survival. Selection chooses the chromosomes to be recombined and mutated out of this initial population. Recombination reproduces the selected individuals and exchanges their information (pair-wise) in order to produce new individuals. This information exchange is called crossover. The crossover procedure takes two selected individuals and combines them about a crossover point thereby creating two new individuals. Mutation introduces a small change to each newly created individual. The resulting individuals are then evaluated through the fitness function. It transfers the information encoded in the chromosome, the so-called genotype, into an execution of the SUT, the so-called phenotype. The fitness function measures how well the chromosome satisfies the test criterion. The implementation of the fitness function follows earlier standards in evolutionary testing, described in other articles [11,12].

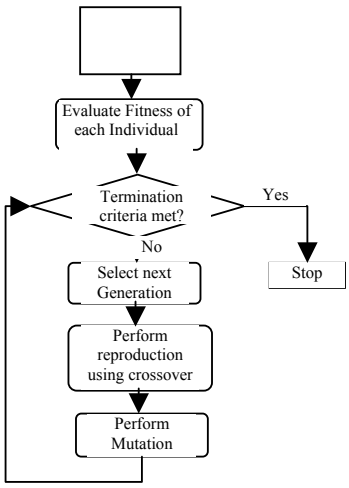

Figure 1. Genetic Algorithm Flow Chart

This iterative process continues until one of the possible termination criteria is met: if a known optimal or acceptable solution level is attained; or if a maximum number of generations have been performed; or if a given number of generations without fitness improvement occur.

III. GENETIC ALGORITHM FOR CLASS TESTING

Object oriented software has more complexity associated with it. Here the objects interact with each other by passing messages. Therefore the representation which is used for procedural software is not capable to represent the object oriented software. One way to deal with the enhanced complexity of objects in evolutionary testing is to enrich the chromosome with representations that are capable to deal with these more complex entities [10]. We need some grammar which can add structure to the chromosome during evolution that can be mapped directly to an executing program. Tonella [10] introduced various notations which can be a base for structuring the chromosomes for applying Genetic Approach to object oriented software.

The genetic algorithms that are based on principle of natural evolution are generalizations of the approach which can be actually implemented by computers are known as Genetic Programming (GP) approach [13]. In our GP approach we represent the test cases as trees [14]. Test cases in object oriented paradigm are basically sequence of method calls. A test case in object oriented software not only contains the numeric test data in form of parameters but also the sequence of constructor and method calls is also necessary [9]. This is because of the following reasons:

1. In a single test case multiple objects may be involved. Therefore the additional objects which may be required as parameters or whose methods may require for Class Under Test (CUT) for execution of its test case. Further it may also be the case that creation of these objects may need to create more additional objects. The set of all these classes from which different class instances are required is called test cluster classes.

2. In order to process any particular test scenario in desired way (e.g. using code coverage criteria certain objects must be in a particular state). Therefore the participating objects are required to be put in those special states to process the test case. Then method calls will be required to be issued for the test cluster objects.

In this way a test case for any object oriented software consists of the definition of testing prerequisites, a test program consisting parameters, their types and their values, method calls as well as the test oracle, which is used for validation of test results. Each test case can be considered as a sequence of statements $S=\{s_1, s_2, \ldots s_n\}$. A statement consists of the following essential components:

- target object
- method
- parameters
- receiver

This is the information which will be needed to encode a genotype individual or chromosome. A method in a statement can be a class method or it may be a constructor. The component parameters will hold the type of the variable (may be primitive or user defined) and their corresponding values. The method which holds the parameters with their types and values are called on the target object in a statement. The component receiver is the variable or object which holds the value which is returned by the method of the target object. Typically any statement s_i can only be executed

if an appropriate target object and all the required parameter objects for its method have been created in advance, i.e. during the calls of s_1 to s_{i-1}. Obviously there exists a call dependency among the methods which must be taken into account when writing statements for test cases. That means any arbitrary sequence of statements is not feasible, but only those which respect the call dependencies.

IV. CHROMOSOME

The GA must be able to generate test programs, but since they don't have understanding of programs, statement or objects therefore some kind of encoding of these components is required to be defined which may allow the representation of a test program as a chromosome which can be used with the GA. The fact that GP is based on hierarchically organized trees requires specialized genetic operators for crossover and mutation. Since we are using trees in order to represent test case sequences, these trees must contain the information of which methods should be called in sequence, and which target objects and parameter objects should be used for the individual method calls. Every operation refers to some object, which is of some type. It also contains some input parameter values as well as their type. These must be created by the GP process and added as leaves to the nodes in the tree representation of the test cases. Each statement s_i is mapped to a sub tree of the entire GP hierarchy, which includes constructors and input values for the required object. The basic primitive types such as boolean, integer, real etc. must also be permitted apart from the user defined object types. These are used primarily to denote input and return values.

While constructing chromosomes the components of each test case statement are encoded using genes. This encoding assigns some number to each corresponding gene. When decoding, the integer values of the genes identify the components such as method to be called and the target object to be used for the invocation. Since the methods in a test cluster usually have parameter lists with different lengths, multiple genes are required to be assigned to represent the parameters for a method. The data type for a parameter gene depends on the parameter to which it is assigned. If the assignment is not clear when encoding, that data type must be used which would allow for every possible decoding. Consequently, a statement from a test program can be represented by a gene indicating the method to be called, a gene indicating the target object for which to call this method, and a number of variables which are used as parameters for the method call. A gene indicating the receiver object is also assigned if it exists in the statement.

All these genes will make a chromosome, which will form the various nodes of the tree representation of test case. For example consider the Figure 2 which shows the test cluster for class Host which is considered CUT here. The test cluster consists of class `Host` and `Connection`.

```
class Host
{
  public Host(Connection con);
  public void connect();
  public void configure(Connection con);
  public Connection getConnection();
  public boolean testPort(Connection con);
  public void send(int[] data);
  public int[] receive();
  public void disconnect();
}
class Connection
{
  public Connection (int port);
  public int getPort();
}
```

Figure 2. A test Class Cluster

The class `Connection` is used as a parameter type of the methods `Host(Connection)` and `Host.configure Connection)` and therefore it is required when we test CUT `Host`. Figure 3 shows an example tree shaped representation of a GP chromosome that translates into the following test case:

```
Connection conn1=new Connection(0x10);
Connection conn2=new Connection(0x20);
Host host=new Host(conn1);
host.configure(conn2);
host.testPort(conn2);
```

The test scenario consists of the creation of two instances of class `Connection` and one instance of class `Host` where one instance of `Connection` is used as the parameter object.

Then `Host.configure(Connection)` is called with second instance of Connection.

Finally `Host.testPort(Connection)` checks the port numbers of two Connection objects whether they are equal or not. If it is, the test passes otherwise it fails.

V. ENCODING

All the components of a test case statement are required to be encoded into genes to form a chromosome so that GA techniques may be applied. The constructor and methods can be encoded by serially numbering to form an ordered set of constructors and methods in the test cluster classes. In the chromosomes a gene is assigned to represent a constructor or method to appear in test program. The domain *D(M)* for the genes can be defined as maximum number of test cluster methods:

$$D(M) = [1,|\ S_C \cup S_M\ |] \subset \mathrm{N}$$

Where $S_C = (c_1,c_2,....c_r)$ and $S_M = (m_1,m_2,....m_n)$ is the ordered set of constructors and methods respectively of the test cluster class. N is the set of all natural numbers.

Similarly a gene must be assigned in the chromosome to identify the target object for which some method will be called by objects of test class cluster. In programming languages this is a kind of object reference based on natural numbers. The domain for the genes of target object is only possible if the number of candidate target objects for a statement s_i is known. This number is directly dependent upon the object creating methods which are called before

statement s_i is called. The domain $D(T_i)$ of target genes for statement s_i can be defined as:

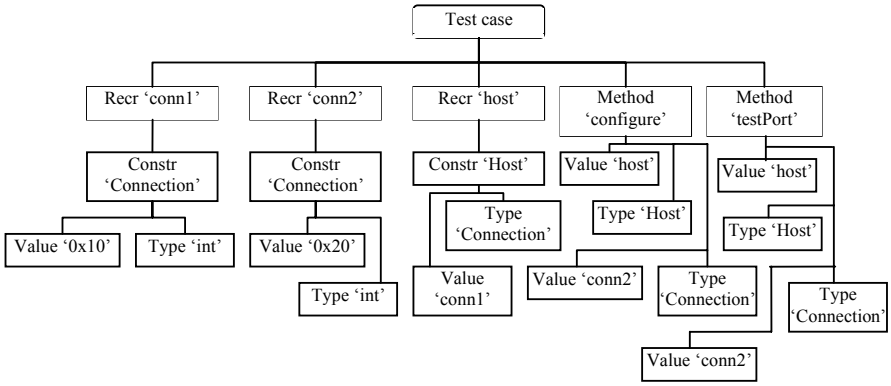

Figure 4. Tree Representation of Chromosome

$$D(T_i) = [1, | O_t^i |] \subset \mathrm{N}$$

where, $t \in C$, the set of all classes in test cluster, is the required target object and i is the index of current statement, $O_t^i = (o_1, o_2, \ldots o_n)$ is the ordered set of objects which are instances of class t and have been created by the statements which are called before s_i.

The encoding of receiver object can be done by assigning a gene for the receiver object used in the statement. The domain of receiver objects can be defined as the objects created by method call in statement s_i.

$$D(R_i) = [1, | O_r^i \cup B |]$$

where r $\in C$, the set of all classes in test cluster, $O_r^i = \{o_1, o_2, \ldots o_r\}$ the ordered set of objects created by method call in statement s_i. B is the ordered set of the basic types available in the language. For each required parameter a gene is assigned in the chromosome. The definition of its domain depends on the data type of the parameter it represents which may be primitive or object type parameters. For primitive data types the domain is same as the data type ranges and precision used in language. For object type parameters a similar object reference mechanism can be used as for the target objects.

VI. Constructing Initial Population

In order to create the first initial population either it may be created randomly or it may be a population based on execution traces. In random initial population creation the initial method invocations as well as their input values are selected randomly. While the second method uses the existing knowledge from executing the SUT. So this way it leads to an initial population that can already cover many of the SUT's runtime paths for a typical usage profiles. Using this method the performance of test generation is increased significantly.

VII. Mutation and Crossover

Mutation is a genetic operator that alters one ore more gene values in a chromosome from its initial state. This can result in entirely new gene values being added to the gene pool. With these new gene values, the GA may be able to arrive at better solution than was previously possible. In GP each individual basic building block requires a separate mutation operator, each of which may be subjected to mutation according to a predefined mutation rate. Three types of mutation operators can be identified, first which creates a new building block, second that changes an existing one and third which deletes a building block. These three operators are devised for each of the components of the test case statement. When these operators are applied to a constructor, then it can be created, deleted or changed to a different constructor. In a similar way normal methods may also be created or deleted. When applied to method parameters their value can be altered. Creation or deletion of a method or constructor will construct or delete its subtree and input parameters also. Figure 4 and 5 show the process of mutation when applied to a tree representation.

In GP the crossover is applied at the nodes which represent genes of the chromosome tree. The nodes over which crossover is applied can be determined randomly, if they are compatible then the crossover operator can be applied and which results in exchanging their subtrees. The compatibility of two nodes is same if they have same type of root node. In simplest way of crossover at the node with method gene the entire method including its input parameters in exchanged.

VIII. Decoding Chromosomes

The chromosome which contains the various genes which represent the various operations as identified. The gene which represents a method can be described by a

function γ which maps each value of method gene to a particular method, associated with it. Suppose $m_k \in S_M$, the kth method in S_M then:

$$\gamma: G_j \rightarrow m_j$$

Where G_j is the genes which was encoded for method m_j.
Now, when decoding for target methods the values must be adjusted to the actual number of candidate target objects. If target T_i was encoded as:
$D(T_i) = R_{Ti}$

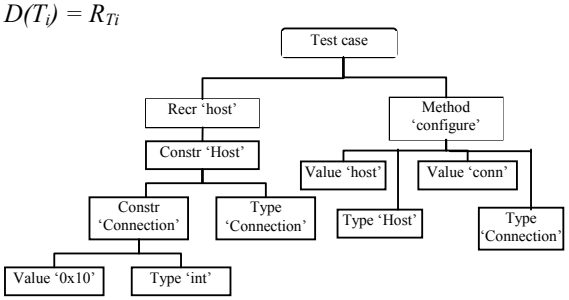

Figure 5. Before Mutation

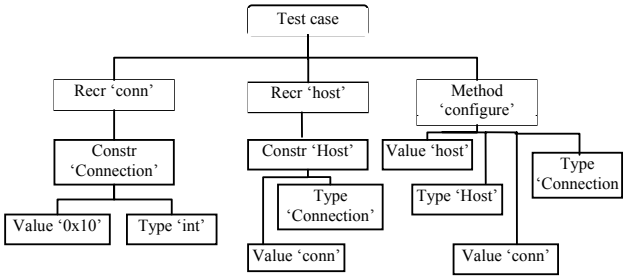

Figure 6. After Mutation

Then decoding can be done by defining a function ρ which assigns each value t of gene G_t , such that $t \in R_{Ti}$ to a target object $o \in O_t^i$:

$$\rho: R_{Ti} \rightarrow O_t^i$$

$$\text{and } t \rightarrow o_t \text{ where } o_t \in O_t^i$$

Similarly when decoding parameter genes, G_p, a function ξ can be defined which assigns each value of G_p which may refer to a primitive value or object reference. If it is a primitive value then no mapping will need, same value can be directly used as parameter value. If the parameter is of object type then a mapping of R_{Pi} to object reference must take place:

$$\xi: R_{Pi} \rightarrow O_p^i$$

$$p \rightarrow o_p$$

where, o_p is the pth object of the ordered set O_p^i .

IX. Objective Function

An objective function is used to guide the algorithm so that it may converge to achieve optimum results. Therefore it assigns each chromosome an objective value which is used to express fitness of the test program [7]. To achieve state coverage keeping the CUT in any specified state we have to test all the methods of the class with its preconditions and post conditions. A variable needs to be defined here which must hold the value which is equal to the difference of total number of methods CUT having and the methods which are tested. Let the total number of methods CUT has is N_M and during any iteration i the number of methods tested is N_E then the objective function must have the following value:

$$d_M = N_M - N_E$$

The goal of objective function is to minimize this value. Since mutation function has a certain probability to change the method parameter. The new values which are assigned should not violate the preconditions of the method. For this purpose we need to define a set of values which will act as allowed ranges for the parameter values. Another parameter which must be included in objective function is d_P which is used to express how close the algorithm has reached in terms of evaluating the methods with representative values and boundary values for each of its parameters list. If N_M represents the total number of methods and then an ordered set of number of parameters for all methods in CUT is $\{x_1, x_2, ... x_{N_M}\}$. Therefore any jth method will have number of parameters equal to x_j. Let the function $\psi(x_j)$ is a boolean value which denotes the complete testing of parameters of jth method. Then d_P must be true, which is defined as below:

$$d_P = \prod_{j=1}^{N_M} \psi(x_j)$$

These factors must be included in objective function and it becomes to minimization of d_M and making d_P true.

X. Experiment

To demonstrate the approach of generating test cases using GP we applied it to some test programs. These test programs were taken from utility package of the open source project HTMLParser. These classes under test included NodeList, NodeIterator, ParserUtils, IteratorImpl, SimpleNodeIterator and other supporting classes. Keeping objective function in view during mutation the probabilities of different operations was set as follows:
Method Introduction: 0.40
Method parameter value alteration: 0.30
Method removal: 0.10
Variable introduction: 0.10
Variable removal: 0.10
The termination criterion for the simulation to finish was making d_M to reach zero and d_P become true. The function $\psi(x_j)$ was defined such that it returns true whenever at least one regular and all boundary values are tested. The result of experiment is shown in Figure 6 from which it is clear that GP approach outperforms as compared to random testing.

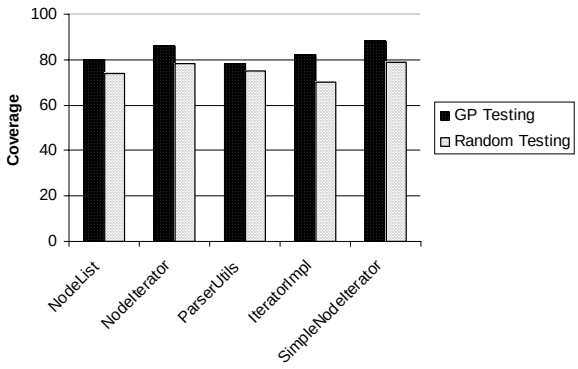

Figure 7.

XI. CONCLUSION AND FUTURE WORK

In this paper, we presented an approach using GP for the generation of test cases. Test cases for testing object oriented software includes test program which create and manipulate objects in order to achieve a certain test goal. The approach described in this paper facilitates the automatic generation of object oriented test program using genetic algorithms. The encoding and decoding of test program into changeable data structures were discussed. Test case generation for the java classes that were introduced in this paper may only be regarded as a proof of the concept and initial step to a more extensive application.

REFERENCES

[1] N. Mansour, and M. Salame, "Data Generation for Path Testing", *Software Quality Control* Vol.12, No.2, pp.121-136, 2004

[2] R.P. Pargas, M.J. Harrold, and R.R. Peck, "Test data generation using genetic algorithms", *The Journal of Software Testing, Verification and Reliability*, Vol.9, No.4, Pp. 263 – 282, 1999

[3] H. Sthamer, "The Automatic Generation of Software Test Data Using Genetic Algorithms", *PhD thesis*, Universityof Glamorgan,Pontyprid,Wales, Great Britain. 1996

[4] B. Beizer, "Software Testing Techniques", 2nd edition, New York: Van Nostrand Reinhold, 1990.

[5] J. Lin, and P. Yeh, "Automatic test data generation for path testing using Gas", *Information Sciences,* Vol.131, No.1-4, pp. 47-64, 2001.

[6] H.-G. Gross, "An Evaluation of Dynamic, Optimisation-based Execution Time Analysis", *International Conference on Information Technology: Prospects and Challenges in the 21st Century (ITPC-2003)*, Kathmandu, Nepal, May 23-26, 2003.

[7] S. Wappler, and J. Wegener, "Evolutionary unit testing of object-oriented software using strongly-typed genetic programming", *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, Seattle, Washington, USA, July 08 - 12, 2006. GECCO '06. ACM, New York, NY, pp. 1925-1932, 2006.

[8] J Wegener and M Grochtmann, "Verifying timing constraints by means of evolutionary testing", *Real-Time Systems*, Vol.3, No.15, pp. 275-298, 1998.

[9] S. Wappler, and F. Lammermann, "Using evolutionary algorithms for the unit testing of object-oriented software". *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, Washington DC, USA, June 25-29, ACM, New York, pp. 1053-1060, 2005.

[10] P Tonella, "Evolutionary Testing of Classes", *Proceedings of the 2004 ACM SIGSOFT Intl. Symposium on Software Testing and Analysis*, Boston, July 11-14, pp. 119-128, 2004.

[11] B Jones et al. "Automatic Structural Testing Using Genetic Algorithms*", Software Engineering Journal*, Vol.11, No.5, 1996.

[12] P McMinn, "Search-Based Software Test Data Generation: A Survey", *Software Testing, Verification and Reliability*, Vol.14, No.2, pp. 105—156, 2004.

[13] D Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA, 1992.

[14] A. Seesing and H.G. Gross, "A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software", *International Transactions on Systems Science and Applications*, Vol.1, No.2, pp.127-134, 2006.