Test Case Generation using GOM Algorithm

Selvakumar Subramanian and Ramaraj Natarajan

***Abstract*—Software testing involves appropriate validation and verification of a software component developed during the software lifecycle. Usually testing costs often account to high budget in the software development process. In order to minimize the testing costs, researchers and practitioners automate the testing process rather than carry out manual testing. Test Case Generation is the process of automatically generating a collection of test cases which are applied to a system under test. This paper utilizes branch coverage criteria using the Generalized Optimization Meta heuristic (GOM) algorithm and code constraint graph (CCG) to efficiently maximize the coverage of all the branches. The experimental results show that the proposed test generation technique is effective in generating tests for an application at large.**

***Index Terms*—Test case generation, branch coverage, evolutionary algorithm, Code Constraint Graph**

I. INTRODUCTION

Testing is the process of exercising a software component using a selected set of test cases, with the intent of revealing defects**.** Testers need to detect these defects before the software becomes operational. Automating the testing process is a relevant issue since it will help reduce analysis costs by enabling a more systematic approach to testing [1]. A good test case is one that has a high probability of revealing a yet undetected defect. It requires the tester to consider the goal for each test case, that is, which specific type of defect is to be detected by the test case. Test Case Generation (TCG) is the process of automatically generating a collection of test cases which are applied to a system under test [28]. White-box TCG is usually performed by means of symbolic execution, i.e., instead of executing the program on normal values (e.g., numbers), the program is executed on symbolic values representing arbitrary values [9]. Test cases should be developed for both valid and invalid input conditions. That is, a tester must not assume that the software under test will always be provided with valid inputs. Inputs may be incorrect for several reasons. For example, software users may have misunderstandings, or lack of information about the nature of the inputs. A test case must contain the expected output or result and the results of the tests should be inspected meticulously. Branch coverage testing criterion encounters all the branches in a program i.e., the predicate of an 'if' statement should be evaluated to both true and false. The stronger criteria of condition, multiple-condition and path coverage are often infeasible to achieve for programs of more than moderate complexity, and thus branch coverage has been recognized as the basic measure for testing.

A small number of test-data techniques have already been automated: random, static and dynamic, analysis-oriented, goal-oriented and structural or path-oriented test-data generators. Random generator is the simplest method of generation techniques, creates large amounts of test data; it could actually be used to generate input values for any type of program. Ultimately, a data type such as integer, string, or heap is just a stream of bits. 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. Since it merely relies on probability it has quite low chances in finding semantically small faults, and thus accomplishing high coverage. A semantically small fault is such a fault that is only revealed by a small percentage of the program input. Static and dynamic generators execute a program symbolically by means of variable substitution techniques instead of actual values. This technique requires plenty of computer resources. It also puts a lot of restrictions on the program. Symbolic execution also implies that a symbolic evaluator for the particular language is built which indeed requires a great amount of work. XML is now being used to replace large relational databases. Therefore, performance testing of XQuery implementations on very large documents is important [4].

Analysis-oriented generators have the ability to generate high quality test-data, but rely upon their designer with a great insight into the domain of operation, and hence are not readily extrapolate to arbitrary software systems. Goal-oriented generators provide guidance towards certain set of paths. Instead of letting the generator generate input that traverses from the entry to the exit of a program, it generates input that traverses a given unspecific path [11]. Because of this, it is sufficient for the generator to find input for any path. Two methods using this technique have been found: the chaining approach and the assertion-oriented approach. The latter is an interesting extension of the chaining approach. Typical for the chaining approach is the use of data dependence to find solutions to branch predicates. The characteristic of chaining is to identify a chain of nodes that are vital to the execution of

Selvakumar. S. is with the Department of Information Technology, Thiagarajar College of Engineering, Madurai, India (Mobile: +91-9789916648; e-mail: sselvakumar@yahoo.com).

Ramaraj N is with the Department of Computer Science &Engineering, G.K.M College of Engineering & Technology, Chennai, India (e-mail: prof.ramaraj@yahoo.in).

the goal node. This chain is built up iteratively during execution. Since this method uses the find-any-path concept it is hard to predict the coverage given as a set of goals. Assertion-oriented testing truly utilizes the power of goal-oriented generation. Certain conditions, called assertions are automatically inserted in the code. When an assertion is executed it is supposed to hold, otherwise there is an error either in the program or in the assertion. But they have serious problems associated with failing to find the global minima. The search space tends to 'lack features' and consists of large 'flat' areas which provide no information on the location or the direction of the true local minima. 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. Structural or Path-oriented generation identifies the path for which the test data is to be generated. Unfortunately, if the path is infeasible that would cause the generator to fail to find an input that will traverse the path. Even though it has the merit of very thoroughly testing a specific path, it has two severe disadvantages. The first is that the number of paths is exponential to the number of branches. The second is that many paths are impossible to exercise due to relationships between the data. Branch coverage criterion measures which decision outcomes of an 'if' statement have been tested. Determining the number of branches in a method is also easy. The total number of decision outcomes in a method is hence equal to the entry branch in the method plus the number of branches that need to be covered.

The rest of this paper is organized as follows. Section 2 discusses research related to the related work. Section 3 presents the details of the proposed approach. Section 4 describes an experimental study of the proposed criterion and observations. Section 5 presents conclusions and future work.

II. RELATED WORK

The work of M.F Bashir, and S.H.K. Banuri, [1] extends the paradigm of the test data generation system to incorporate both specifications and model based testing which helps to perform the reclassification of the code, specification or model based techniques. Several attempts have been made to develop a system to generate test data automatically. The existing such system does not guarantee to generate test data in only feasible paths. Praveen Ranjan Srivastava et al. [2] proposed a method to generate feasible test data, using Genetic Algorithm. It is often desired that test data in the form of test sequences within a test suite can be automatically generated to achieve required test coverage. The work of Sushil K. Prasad et al. [3] proposes Genetic algorithm to test data generation for optimizing software testing. Ana Barbosa et al. [5] proposed a test case generation approach to model-based testing of graphical user interfaces from task models. [5] shows how task mutations can be generated automatically, enabling a broader range of user behaviors to be considered. More recently, Grammar-based test generation has been applied to many other testing problems, including the generation of eXtensible Markup Language (XML) documents and the generation of packets for testing communications protocols [6].

Recent research has shown how to integrate covering-array techniques such as pairwise testing into Grammar-based test generation tools [6]. Their work proposed two case studies showing how to use grammars and covering arrays for automated software testing. Valentin Dallmeier et al. [7] combined systematic test case generation and typestate mining, static typestate verifier fed with enriched models report significantly more true positives, and significantly fewer false positives than the earlier proposed models. [8] proposed an automatic test generation solution using dynamic symbolic execution, uses distance in control-dependency graph to guide path exploration towards the change. [8] is effective in generating change-exposing inputs for real-world programs. [9] proposed a symbolic execution mechanism, by developing a fully Constraint Logic Programming based framework for test case generation of an OO imperative language. Rafael Caballero et al. [10] presented a general framework for generating SQL query test cases using Constraint Logic Programming. [11] presented an automated approach to generate unit tests that detect these mutations for object-oriented classes, the resulting test suite is optimized towards finding defects rather than covering code and the state change caused by mutations induces oracles that precisely detect the mutants. [12] proposed an approach for automated test case generation based on techniques from constraint programming. [13] proposed a scalable toolset using Alloy to automatically generate test cases satisfying T-wise from SPL models. [13] defines strategies to split T-wise combinations into solvable subsets.

III. OVERVIEW OF THE APPROACH

A. Framework

The block diagram of this proposed approach is depicted as in Fig.1. It consists of a three-tier architecture containing the following four blocks: Source code analyzer, XML parser, Constraint analyzer and Test data generator. Initially, a sample code consisting of only 'if-else' constraints is taken as input. As the name 'constraint' specifies, the input code that is to be tested should never contain or be allowed any loops such as 'while' or 'for' loops. The *source code analyzer* block analyses the input code and generates an XML document which separates the constraints and their outcomes and also neglecting the statements if any, present. The *XML parser* block initiates the XML document and creates a notepad file containing the constraints similar to those present in the XML document, given as the input file to the algorithm employed i.e., the evolutionary meta-heuristic algorithm. The primary portion of the third block which is the *constraint analyzer* takes the notepad file generated earlier as input and generates a graph called the Code Constraint Graph (CCG). This graph indicates the program flow of the source code as to which branches are present and which are to be tested. The CCG is no longer used because it just shows the control flow of the input source code. The secondary portion of the third

block called the *test data generator* block implements the evolutionary algorithm namely the Generalized Optimization Meta-Heuristic (GOM) algorithm, which takes the notepad file to be checked against the constraints separated.

The test data generator block first generates a random set of integers which may range from positive to negative. The set may contain an approximate number of 40-50 random integers. For each of the random set of integers, a fitness formula is evaluated in order to advance to the next population. After calculating the fitness for each of the random set of integers, fitness values are assigned to them. The chromosome with the optimal fitness is chosen as the base chromosome for the next generation of members. Each chromosome contains 24 bits. So the 24 members of the next population are generated from the base chromosome by flipping each bit of it. Those newly generated members are then calculated by the given fitness functions. Then the chromosomes are ranked according to their fitness values, from worst to the best. The optimal chromosomes of the next and the previous population are compared. The population that has a comparatively lesser optimal chromosome is deleted and the other one is kept as the base chromosome for the generation of the members of the next population. Cross over and mutation operations are applied to them if necessary.

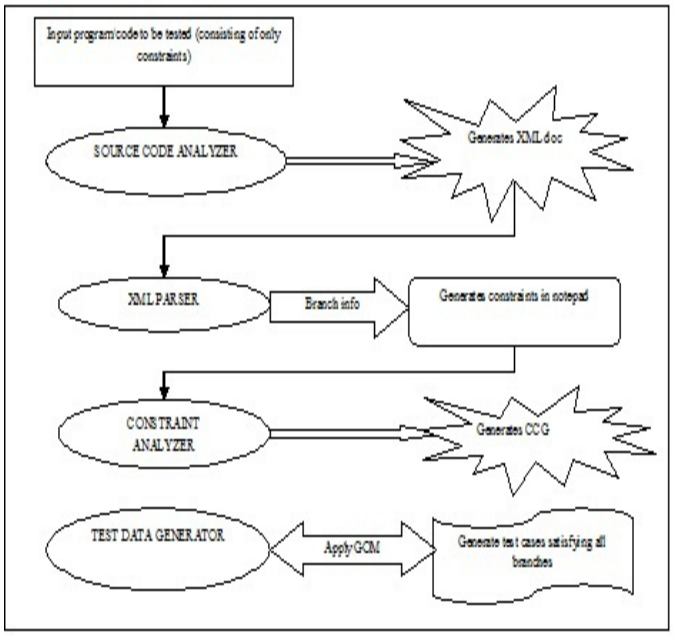

Fig. 1. Proposed Framework for test case generation

B. Motivational Example

Consider the triangle classification program in Fig. 2, which accepts three variables say A, B and C each for the three sides of a triangle. The triangle classification accepts three integers as the three sides of a triangle, and decides which type of triangle it based upon the length of these three slides. The four possible results are: scalene, isosceles, equilateral and not a triangle. If all the three sides are equal, the program returns an equilateral triangle. If any two sides are equal, the program returns an isosceles triangle. If none are equal, then the program returns a scalene triangle. So in this algorithm the above procedure of ranking and cross-over mutation operations are repeated three times, i.e., each for three sides of a triangle A, B, C. The generated members of the first population contain the chromosomes each with three genes that indicate the three variables used in that program. Every chromosome is then passed to the notepad file and checked against the constraints. When the chromosomes passed satisfy the particular constraints, a counter variable is incremented so as to count the number of branches satisfied by them. This procedure is repeated for each chromosome of the first population. After completing all the members of the first population, the same operations are performed for each pair of genes and then by applying cross over and mutation operations. The above procedure is repeated till the maximum number of branches is satisfied by the branch coverage criterion. The GOM algorithm generates three genes in each chromosome, as the number of variables involved in the source code is three. Before the generation of the appropriate genes in a chromosome, the program module is first parsed and all the 'if-else' branches involved are separated in a notepad file in order to easily give input to the GOM algorithm.

```
int triType(int a, int b, int c) {
 1   int type = PLAIN;
 1   if (a < b)
 2     swap(a, b);
 3   if (a < c)
 4     swap(a,c)
 5   if (b < c)
 6     swap(b, c)

 7   if (a == b) {
 8     if (b == c)
 9       type = EQUILATERAL;
 .     else
10       type = ISOSCELES;
 .   }

11 else if (b == c)
12   type = ISOSCELES;
13 return type;
}
```

Fig. 2. Module of triangle classification.

C. Formulation of branch coverage

The proposed approach in this paper utilizes branch coverage criteria using a Generalized Optimization Meta heuristic (GOM) algorithm and code constraint graph (CCG). The process of generating test cases using GOM algorithm and CCG graphs is as follows; assume the input that is to be checked against the constraints in the source code to range *I* as: $\{I_1, I_2, \ldots, I_n,\}$. Define and identify the test constraint set C as: $\{C_1, C_2, \ldots, C_m\}$. For each and every input I, the test cases (*Tc*) are generated so that those inputs must satisfy the appropriate constraints encountered in the constraint set and this process is repeated until the maximum branch coverage is attained as output for the given input set. The amount of branch coverage (T_{bc}) criterion can be expressed mathematically as:

$$T_{bc} = (BC + SC + MC / 2 \times B + S + M) X\ 100$$

$$FITNESS = 100 - T_{bc}$$

Where *BC* - number of branches covered
SC - number of statements covered
MC - number of methods covered

B – Total number of branches
S – total number of statements
M – total number of methods

The steps to be employed are as follows:

- A sample input code containing 'if-else' constraints is taken as input.
- The constraints present in the source code are separated by reading line by line and the unnecessary statements, header files, braces and new lines are removed.
- The GOM algorithm is employed after generating a random set of integers say, up to 40-50 iterations.
- The test cases are generated by the GOM algorithm indicating which branches are covered by appropriate chromosomes, each containing three genes.
- The percentage of branch coverage by the chromosomes is obtained by applying the relevant formulas regarding the branch coverage.
- The highest percentage of coverage by the chromosomes is returned as the best solution.

D. Proposed pseudo code of the algorithm

Fig. 3 depicts the pseudo code of a Generalized Optimization Meta-Heuristic (GOM)

Procedure TDGen
Input:
Program: code/program under test
Output:
CCG: A code constraint graph, which shows the control flow of the source code.
Test cases: A set of test cases that is generated using GOM algorithm.
begin

1. A sample code consisting of "if-else" conditions is taken as input.
2. The source code analyzer generates an XML document by analyzing the constraints in the source code and separates them in that document.
3. The XML parser goes through the XML document created and creates a notepad file containing only constraints so as to easily give as input against the GOM algorithm.
4. The constraint analyzer accepts the notepad as input and generates a tree called Code Constraint Graph (CCG) so that it shows a program flow of the source code with constraints as which branches are present and that are to be tested.
5. For each entry of the requirements. Initialize randomly the population of S species (bits) that are used to encode D program variables.

loop

- *For each of the S bits of the species, find the fitness value.*
- *Find the base chromosome that has the best fitness among the generated members.*
- *Generate first population using bit changes to the base chromosome, as much times as its bits.*
- *Evaluate the first population against the fitness function to find the fitness value, and assign to each of the members.*
- *Perform ranking operation to rank the members according to their fitness value.*
- *Apply either "one-point" or" two point" cross over operation by taking either best and its successor or worst and its predecessor.*
- *Modify a single bit of the gene with probability* $P \alpha k^{-\mu}$ *(where k is the rank of the bit and μ, a free control parameter)*
- *Compare the first population and next population fitness values. Discard the population which has worst base chromosome.*
- *Repeat the above procedure for next population.*
- *Update branch coverage criteria, test cases and iteration counter.*

until stopping criterion is met

6. Return the branch coverage result and test cases.

end

Fig. 3 Generalized Optimization Meta-Heuristic (GOM) pseudo code

Separation of constraints from the source code: The source code itself cannot be tested since it has irrelevant codes such as printing statements, other logical statements, etc. So it is necessary to separate the constraints alone from the source code, in this case the branches with their appropriate outcomes i.e., children. The methodology used to separate the branches alone from the source code consists of reading each and every line of the source code until a branch or constraint is encountered. A better way to implement this methodology is to read the source code and place them in an XML document. As seen in Fig. 1, the source code analyzer block performs this operation. As the branches are separated in that XML document, the unnecessary new lines, braces and other header files are removed so that only the constraints are filtered. The generated XML document is parsed and the appropriate operations are performed so that the unnecessary codes are removed. As seen in Fig.1, the XML parser block performs this operation. The filtered branches are placed in a notepad file, given as input to the GOM algorithm. In the notepad, an @ symbol is referenced before each branch so that a constraint is encountered. If any children's are present in a branch, the @ symbol is incremented and if the line comes outside the 'if-else' branch, the @ symbol is appropriately decremented. The generated filtered branches and/or constraints from the source code are visualized as in Fig. 4 as:

```
enter tritype
@if(a<b)
@@swap(a,b);
@if(a<c)
@@swap(a,c);
@if(b<c)
@@swap(b,c);
@if(a==b)
@@if(b==c)|
@@@type=eqilateral;
@@else
@@@type=isoceles;
@elseif(b==c)
@@type=isocesles;
@return type;
```

Fig. 4 Sample constraint generation

Generation of Code Constraint Graph (CCG): The Code Constraint Graph can be generated from the notepad file created as in Fig.3 to show the control flow of the source code consisting of constraints. As in fig. 1, the constraint analyzer block performs this operation of generating the CCG. The sample CCG generated from the triangle classification module can be visualized as in fig. 5 as shown below:

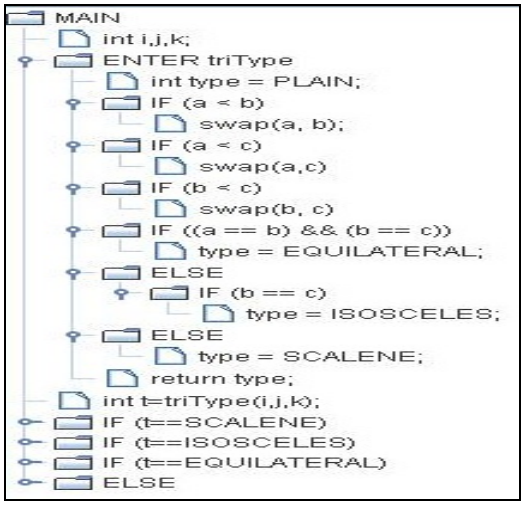

Fig. 5. CCG generation

Generating test cases using proposed evolutionary algorithm: The proposed Generalized Optimization Meta Heuristic (GOM) algorithm is a type of an evolutionary algorithm (EA) that makes use of evolutionary [16, 17] strategies (ES) and evolutionary programming (EP). The main idea behind the EAs is to evolve a population of individuals (candidate solutions for the problem) through competition, mating and mutation [18], so that the average quality of the population is systematically increased in the direction of the solution of the problem at hand. The evolutionary process of the candidate solutions is stochastic and "guided" by the setting of adjustable parameters. In an analogy with a natural ecosystem, in a EA different organisms (solutions) coexist and compete. The more adapted to the design space will be more prone to reproduce and generate descendants. On the other hand, the worst individuals will have fewer or no offspring. In an optimization problem, the fitness [19, 20] of each individual is proportional to the value of the objective (cost) function, also called fitness function.

In a GOM algorithm, each bit is considered a species and a string of S bits is taken as the initial population of the species. The string consisting of S bits then encodes the D program variables to be represented in a binary format of 0's and 1's. In a variation of the canonical GOM described above, the bits are ranked separately for each substring that encodes each program variable, and N bits, one for each variable are flipped at each iteration of the algorithm. First, the GOM algorithm generates the random integers of up to 40-50 numbers. From the generated random numbers, each and every two pair of integers is taken into account. The second number of the pair is taken and the bits of that number are ranked according to their priority. The highest and the lowest bits are taken as an average to obtain a result. In the first number of the pair the shifting operation is performed, as many times as the result of averaging. The same procedure is repeated for the second number and the modified first and second numbers are kept aside. Then these numbers undergo appropriate cross over and mutation operations. The cross over method used here is the "two point" cross over.

The mutation operation is then performed with respect to the probability $P \alpha k^{-\mu}$, where k is the rank of the bit and μ, a free control parameter. These optimized integers are then checked against the notepad file generated earlier. When the chromosomes satisfy a particular branch, a counter variable is incremented which indicates the number of branches that are satisfied by a single chromosome. In this case, a single chromosome consists of three genes since the number of variables encountered in the triangle classification is three namely A, B, C. The above procedure is repeated till the chromosomes in all populations by means of branch coverage criteria cover a maximum number of branches. The general representation of two-point cross over can be represented as: The two point crossover operator takes two vectors $(a_1,\ldots, a_n)$ and $(b_1,\ldots, b_n)$ and yields two vectors $(a_1,\ldots, a_i, b_{i+1},\ldots, b_j, a_{j+1}, \ldots\ldots, a_n)$ and $(b_1,\ldots, b_i, a_{i+1},\ldots, a_j, b_{j+1}, \ldots\ldots, b_n)$, where $1 \leq i < j \leq n-1$ and i and j are randomly chosen. This means that both vectors are split at the same two positions and assembled with swapped middle parts. An example of a mutation operation performed is, Before: 1 1 0 1 1 0 1 0 0 1 1 0 1 1 1 0, After: 1 1 0 1 1 0 0 0 0 1 1 0 1 1 1 0, a change of bit in the gene takes place at bit position 6.

IV. THE EXPERIMENT

A. Subject Programs, Faulty Versions, and Test Case Pools

To examine the efficacy of our approach, the proposed approach was evaluated using real-world programs. In this section, we report the empirical evaluation results. We compared the GOM Coverage with the GA Coverage. In the experiments, the Siemens suite programs (Table 1), similar to those used by Dawei Qi et al. [8] and Rothermel et al. [15] were used to validate the performance of the proposed approach. Each program was hand-instrumented to record all the coverage information. Each program has a variety of

versions, each containing one fault. Each program also has a large universe of inputs. We obtained the subject programs from the Software-artifact Infrastructure Repository at UNL [14].

TABLE I
SIEMENS SUITE SUBJECT PROGRAMS

Name	Lines of code	Faulty version count	Test pool size	Program Description
tcas	162	41	1608	Altitude separation
totinfo	346	23	1052	Information measure
schedule	299	9	2650	Priority scheduler
schedule2	287	10	2710	Priority scheduler
printtokens	378	7	4130	Lexical analyzer
printtokens2	366	10	4115	Lexical analyzer
replace	514	32	5542	Pattern replacement
Space	9127	38	13,585	Array definition language interpreter

B. *Experimental Results and Observations*

To examine the efficacy of our approach, we evaluated our approach using real-world programs. In this section, we report our empirical evaluation results. The obtained results of branch coverage criteria can be depicted by a graph as in Fig. 6 showing the convergence of coverage. The number of iterations is scaled along the X-axis and the percentage of branch coverage is scaled along the Y-axis. As compared to the simple genetic algorithm, GOM algorithm converges faster in less number of iterations. The maximum branch coverage obtained by applying the proposed algorithm is nearly 71%. As seen in Fig. 6, the applied GOM algorithm converges at a faster rate than the simple genetic algorithm i.e., at iteration 70 (number 7), the GOM reaches the maximum branch coverage of 71% and it is consistent in the further numbers of iterations, whereas the genetic algorithm reaches the maximum branch coverage of 64% only at iteration 100 (number 10).

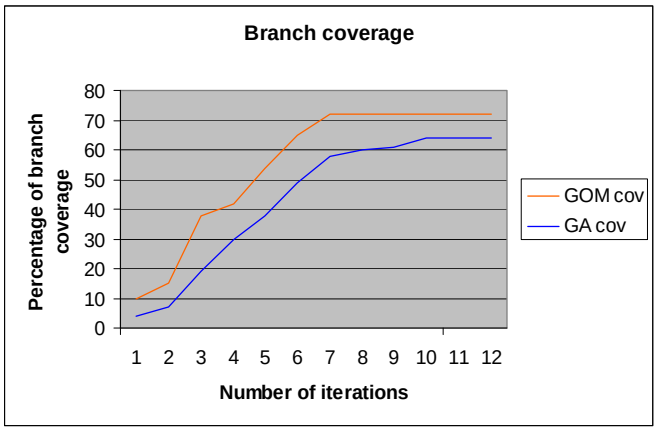

Fig. 6. Comparison of branch coverage of GA and GOM

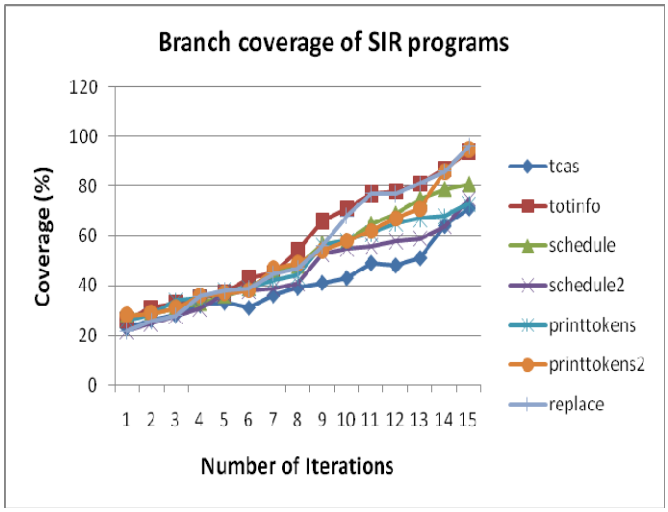

Fig. 5. Branch coverage of SIR objects

Fig. 7 shows the Branch coverage of the SIR objects. In terms of numbers, the vast majority of all test cases have at least one assertion. Although the achieved score is quite high, the search based approach offers potential for optimization. While the coverage based impact measurement guides the search towards assertions, in the experimental of SIR programs, for certain cases GOM is fair in coverage.

V. CONCLUSIONS AND FUTURE WORK

The GOM algorithm implemented gives a suitable way for automatic test case generation. The ease of test case generation is faster than with the simple genetic algorithm since the number of iterations for reaching the optimal solution is quick. The separation of constraints from the source code and then exporting them to a separate notepad file makes the implementation of this algorithm further easier. The code constraint graph (CCG) generated allows understanding the control flow of the source code to depict the amount of statements, branches and methods present and also which are covered. The cross-over and mutation operations are optional, since the algorithm has the capability to converge well without performing those operations. In terms of future work, we can extend our method by improving the fitness function to

deduce a better result above the maximum amount of coverage obtained.

ACKNOWLEDGMENT

We thank Dr. Gregg Rothermel, Dept. of Computer Science, University of Nebraska, for providing the Siemens Suite of programs.

REFERENCES

[1] Bashir, M.F, Banuri, S.H.K., "Automated model based software Test Data Generation System", Proc. 4th International Conference on Emerging Technologies, pages 275-279, Oct.2008.

[2] Praveen Ranjan Srivastava, Priyanka Gupta, Yogita Arrawatia, Suman Yadav, Use of genetic algorithm in generation of feasible test data ACM SIGSOFT Software Engineering Notes, Vol. 34,Issue 2, March 2009

[3] Sushil K. Prasad, Susmi Routray, Reema Khurana and Sartaj Sahni, "Optimization of Software Testing Using Genetic Algorithm" Third International Conference Information Systems, Technology and Management, pages 350-351, March 2009

[4] Daniel Homan, David Ly-Gagnon, Paul Strooper, and Hong-Yi Wang, Grammar-Based Test Generation with YouGen, Software Practice And Experience, 1:1-20, 2009.

[5] A. Barbosa, A. Paiva and J.C. Campos, Test case generation from mutated task models, ACM Symposium on Engineering Interactive Computing Systems, 2011.

[6] Daniel Hoffman, Hong-Yi Wang, Mitch Chang, David Ly-Gagnon, Lewis Sobotkiewicz, Paul Strooper, Two case studies in grammar-based test generation, Journal of Systems and Software, Volume 83 Issue 12, December, 2010.

[7] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, Andreas Zeller, "Generating test cases for specification mining", Proceedings of the 19th international symposium on Software testing and analysis ACM, NY, USA 2010.

[8] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, Test Generation to Expose Changes in Evolving Programs, 25th IEEE/ACM International Conference on Automated Software Engineering (ASE), September 2010.

[9] Miguel G´Omez-Zamalloa1, Elvira Albert1, Germ´ An Puebla, Test Case Generation for Object-Oriented", Imperative Languages in CLP, Journal of Theory and Practice of Logic Programming, Volume 10 Issue 4-6, July 2010.

[10] Rafael Caballero, Yolanda García-Ruiz, Fernando Sáenz-Pérez, " Applying Constraint Logic Programming to SQL Test Case Generation", Proceedings of 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010.

[11] Gordon Fraser, Andreas Zeller, "Mutation-driven Generation of Unit Tests and Oracles" Proceedings of the 19th international ACM symposium on Software testing and analysis, NY, USA, 2010.

[12] François Degrave, Tom Schrijvers and Wim Vanhoof, "Towards a Framework for Constraint-Based Test Case Generation", Logic-Based Program Synthesis And Transformation, Lecture Notes in Computer Science, 2010, Volume 6037, 2010.

[13] Perrouin, G., Sen, S., Klein, J., Baudry, B., le Traon, Y., "Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines" , Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, Paris, pages 459 - 468, 6-10, April 2010.

[14] Software infrastructure repository (SIR) http://www.cse.unl.edu/~galileo/sir.

[15] Gregg Rothermel,M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. International Conference on Software Maintenance, pages 34–43, November 1998.

[16] S. Horwitz. Tool support for improving test coverage. In Proceedings of ESOP 2002: European Symposium on Programming, 2002.

[17] P. McMinn, M. Holcombe. The State Problem for Evolutionary Testing, GECCO, 2003, pp. 2488–2498.

[18] C.C. Michael, G. McGraw, M. Schatz. Generating Software Test Data by Evolution, IEEE transactions on software engineering 27 (12) (2001) 1085–1110.

[19] P. Thevenhod-Fosse, H. Waeselynck, STATEMATE: applied to statistical software testing, Proceedings of the 1998 International Symposium on Software Testing and Analysis, 1998.

[20] J.M. Voas, L. Morell, K.W. Miller, Predicting where faults can hide from testing, IEEE Software 8 (2) (1991) 41–48.

Selvakumar. S is completing his Ph.D research work in Computer Science & Engineering from the Anna University. He received the Masters Degree in Computer Science & Engineering from Madurai Kamaraj University in 1996. He received the Master of Business Administration from the same University. He has over 15 years experience in various institutions and Organizations. Currently he is an Assistant Professor in the Department of Information Technology, Thiagarajar College of Engineering. He has long been interested in Software Engineering. His research interests include Software Testing, Software Quality Engineering, Software Project Management, Software Metrics and Data mining, Data Base Systems. He also had a carrier as a developer for real-time, business-critical systems. Thus he has experience both with the practical problems of software development and the theoretical underpinnings of Software Engineering and Computer Science. He has presented a number of papers in international journals and in various other journals. He has carried out various sponsored Short Term Training Programs and worked on various Projects. He is a Senior Member in the Computer Society of India, Member in the Institute of Engineers and the Indian society of Technical Education.

Ramaraj. N received bachelor degree in Electrical and Electronics Engineering from Madurai Kamaraj University, Madurai, Tamil Nadu, in the year 1976. Received Masters Degree in Power System Engineering from the Madurai Kamaraj University, Madurai, Tamil Nadu, in the year 1980. Received PhD degree from Madurai Kamaraj University, Madurai, Tamil Nadu in the year 1992 in Computer Applications. He is the Principal of GKM Engineering College, Chennai, Tamil Nadu, India. He has more than 25 years of teaching experience and has presented a number of papers in international journals (20) and in various other journals (35). His area of interest is Artificial Intelligence, Software Engineering, Software Testing and Distributed Computing. He is a Member in the Institute of Engineers and the Indian society of Technical Education.