ISSN: 2319-7080 International Journal of Computer Science and Communication Engineering
Volume 2 Issue 3 (August 2013 Issue)

Design and Implementation of Testing tool using Genetic Algorithm for Test Case Prioritization

Kirandeep Kaur[1], Vinay Chopra[2]

[1]M.Tech Scholar,KC College of Engineering and Information Technology, Nawanshahr (Punjab), India

[2]Assistant Professor, DAV Institute of Engineering & Information Technology, Jalandhar (Punjab), India

[1]er.kiran2007@gmail.com

***Abstract*: Software life cycle does not get complete without the proper testing of the software because a software performance gets decreased when we don't minimize the small faults present in the code. A lot of testing techniques have been already implemented in the same contrast like Black box testing, white box testing and others. This paper focuses the improvement of the quality on the basis of the releasing unused memory when the software runs. There are always some variables in the code which are not used at run time and engage the software in producing more time to produce the output. In addition to the variables there are functions also which are not used but they block some memory in the lifecycle of the software. The computation of these unused memory can be done using Genetic Algorithm which passes the objective function a list of those blocks which are not used and the fitness function computes that what exact memory space can be freed and how the software can release the memory. For this task to achieve we have used visual studio as front end and atcut matrix dynamic link libraries for the comparison.**

***Keywords* - Software Memory, Genetic Algorithm, ATCUT matrix**

I. INTRODUCTION

The software testing is designed with the purpose of detecting defects for a given set of inputs and estimating the operational effectiveness and suitability of the program being developed. Software program could be viewed as a function that describes the relationship of an input to an output. The testing process is used to ensure that the program realizes the function and its problem could be represented by the testing problem. The essential components of a software program test are a description of the functional input range, the program in executable form, a description of the expected behavior, a way of observing program behavior and a method of determining whether the observed behavior conforms to the expected behavior. Generally, software-testing techniques are classified into two categories: static analysis and dynamic testing [1].

II. SOFTWARE TESTING TECHNIQUES

There are following software testing techniques such as:

- **Black Box Testing :**
  The technique of testing without having any knowledge of the interior workings of the application is Black Box testing. The tester is oblivious to the system architecture and does not have access to the source code. Typically, when performing a black box test, a tester will interact with the system's user interface by providing inputs and examining outputs without knowing how and where the inputs are worked upon.

- **White Box Testing :**
  White box testing is the detailed investigation of internal logic and structure of the code. White box testing is also called glass testing or open box testing. In order to perform white box testing on an application, the tester needs to possess knowledge of the internal working of the code [2].The tester needs to have a look inside the source code and find out which unit/chunk of the code is behaving inappropriately.

- **Grey Box Testing:**
  Grey Box testing is a technique to test the application with limited knowledge of the internal workings of an application. In software testing, the term *the more you know the better* carries a lot of weight when testing an application.

  Mastering the domain of a system always gives the tester an edge over someone with limited domain knowledge. Unlike black box testing, where the tester only tests the application's user interface, in grey box testing, the tester has access to design documents and the database. Having this knowledge, the tester is able to better prepare test data and test scenarios when making the test plan.

III. PREVIOUS APPROACH USING GENETIC ALGORITHM

One of the major difficulties in software testing is the automatic generation of test data that satisfy a given adequacy criterion. To solve this difficult problem there were a lot of research works, which have been done in the last 19 years. Perhaps the most commonly encountered are random test-data generation, symbolic (or path-oriented) test-data generation, dynamic test-data generation and recently, test-data generation based on genetic algorithms (GAs). Xanthakis et al. in [3] is presented the first work applying genetic algorithms to generate test data. In this work GAs are employed to generate test data for structures not covered by random search. A path is chosen by the user and the relevant branch predicates are extracted from the program. The GA is then used to find input data that satisfies all branch predicates at once, with the fitness function summing branch distance values. Alan Schultz et al [4] in 1993 propose a machine learning techniques to evaluate autonomous vehicle software controller. A set of simulated fault scenarios is applied to controller and a genetic algorithm searches for significant combination of fault. This approach to find a minimum set of faults that produces degraded vehicle performance and maximum set of faults that can be tolerated without significant performance loss. Pei et al. [5] in 1994 observed that most of the test data generators, which were developed in their era, were using symbolic evaluation. They observed that both static and gradient-descent-based dynamic testing was developed. However, they concluded that static testing was not practical, while the dynamic one was not effective. These drawbacks had inspired Pei et al. to develop a single-path-coverage test data generator that employs GA. Hunt J. et al. [6] 1995 presents a genetic algorithm designed to search for significant input and output combinations to software control system. By "significant" is meant those which produce an output (or result) which is not in line with the original specification. It is intended that such a tool should be used to support the human tester by focusing their attention on areas of concern which they can investigate further. Around the same time, Roper et al. [7] in 1995 developed a GA based test data generator that has an aim to traverse all the branches within a target program. Their generator takes a program and instruments it automatically with probes to provide feedback on the branch coverage achieved. Alander, J. et al [8] in 1996 studying possibilities to test software using genetic algorithm search. The idea is to produce test cases in order to find problematic situation like processing time extremes. The proposed test method comes under the heading of automated dynamic stress testing. One year after, Jones et al. [9] in 1996 developed a similar GA based test data generator to achieve branch coverage. Their major contributions are the use of a sequence of binary strings for individual representation, which is converted to a decimal number prior to the program execution and the use of unrolled control flow graph (CFG) to represent one, two, or more iterations for each loop, which makes the CFG acyclic. As each branch is executed, the test data generator automatically traverses the CFG to the next branch in a breadth-first manner. Michael et al. [10] in 1997 implemented Corel's function minimization approach [21] in their GA-based test data generator. They have built a test data generator called GADGET (Genetic Algorithm Data Generation Tool), which has the ability to instrument a program automatically with no limitation in the programming language, but it has a restriction that it can only accept scalar inputs. GADGET has the condition-decision coverage as its adequacy criteria. GADGET uses simple GA as well as differential GA. The difference between differential GA and the simple GA is in the recombination process [21]. Michael et al.'s result shows that, in general, the simple GA outperforms the differential one. GADGET is considered to be the first test data generator to be tested against a large real-world program named b737, which is part of an autopilot system (real-world control software). Michael et al. reported that the performance of random test generation deteriorates for larger programs.

IV. GENETIC ALGORITHM

Genetic Algorithms begins with a set of initial individuals as the first generation, which are sampled at random from the problem domain. The algorithms are developed to perform a series of operations that transform the present generation into a new, fitter generation [22].

Each individual in each generation is evaluated with a fitness function. Based on the evaluation, the evolution of the individuals may approach the optimal solution.

The most common operations of genetic algorithms are designed to produce efficient solution for the target problem [15]. These primary operations include:

- **Reproduction-** This operation assigns the reproduction probability to each individual based on the output of the fitness function. The individual with a higher ranking is given a greater probability for reproduction. As a result, the fitter individuals are allowed a better survival chance from one generation to the next.

- **Crossover-** This operation is used to produce the descendants that make up the next generation. This operation involves the following crossbreeding procedures:
  - Randomly select two individuals as a couple from the parent generation.
  - Randomly select a position of the genes, corresponding to this couple, as the crossover point. Thus, each gene is divided into two parts.

- Exchange the first parts of both genes corresponding to the couple.
- Add the two resulted individuals to the next generation.

- **Mutation-** This operation picks a gene at random and changing its state according to the mutation probability. The purpose of the mutation operation is to maintain the diversity in a generation to prevent premature convergence to a local optimal solution. The mutation probability is given intuitively since there is no definite way to determine the mutation probability [22].

Methodology

The approach is quite simple and effective.If the mutation changes we need to check how much time has been consumed and whether the memory cycle has changed or not .If the memory cycle changes then it indicates that unused memory has been introduced through the block of program and fitness function indicates the location of the unused memory. The performance can be measured on the basis of the following parameters:

- **RFC (Response for Class):** It indicates the response which has been given from a code block when it is called to be executed.
- **NOC (Number of Child Nodes):** It basically represents the inheriting structure of the class. More number of classes inherited into one single block, less amount of memory it would occupy.
- **Encapsulation:** It encapsulates the result into percentage time found dead in waiting the response from a class.

V. ATCUT AND ITS MATRICES

An Atcut matrix is combinations of several parameters on the basis of which we have choose to perform our task analysis. Rather than involving the previous matrices parameters like software tolerance, maintainability it involves new parameters which supports the extended programming of any platform. Atcut matrices has been composed in such a manner that each and every section of the software gets covered and each section can be evaluated on the basis of the code written for it . It involves the following parameters:

- **Depth of Inheritance tree (DIT):** It is "the maximum length from the node to the root of the tree". ***Theoretical basis*** it gives a measure of ancestor classes that can potentially affect this class. Higher value of DIT shows a higher potential for reuse but increased complexity.
- **Number of children (NOC)***:* It is the number of immediate sub-classes subordinated to a class in the class hierarchy. ***Theoretical basis*** It is a measure of the subclasses that are going to inherit the methods of the parent class. HighNOC value indicates high potential for reuse and likelihood of improper abstraction.

- **RFC (Response for Class):** The RFC is the "Number of Distinct Methods and Constructors invoked by a Class." The response set of a class is the set of all methods and constructors that can be invoked as a result of a message sent to an object of the class. This set includes the methods in the class, inheritance hierarchy, and methods that can be invoked on other objects.

  The RFC for a class should usually not exceed 50 although it is acceptable to have a RFC up to 100. RefactorIT recommends a default threshold from 0 to 50 for a class. If the RFC for a class is large, it means that there is a high complexity. For example, if a method call on a class can result in a large number of different method calls on the target and other classes. It can be hard to test the behavior of the class and to debug problems since comprehending class behavior requires a deep understanding of the potential interactions that objects of the class can have with the rest of the system

- **Encapsulation:**In computer networking, **encapsulation** is a method of designing modular communication protocols in which logically separate functions in the network are abstracted from their underlying structures by inclusion or information hiding within higher level objects.

VI. RESULTS

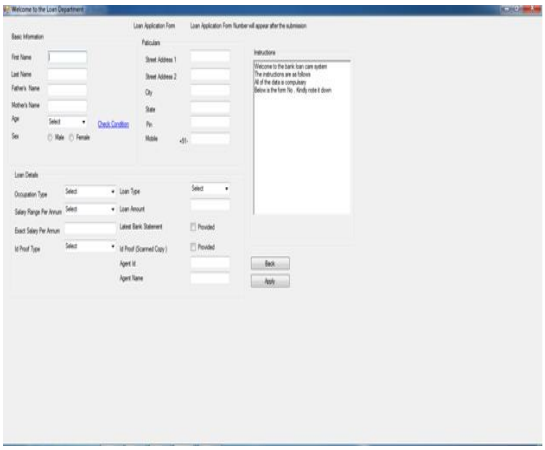
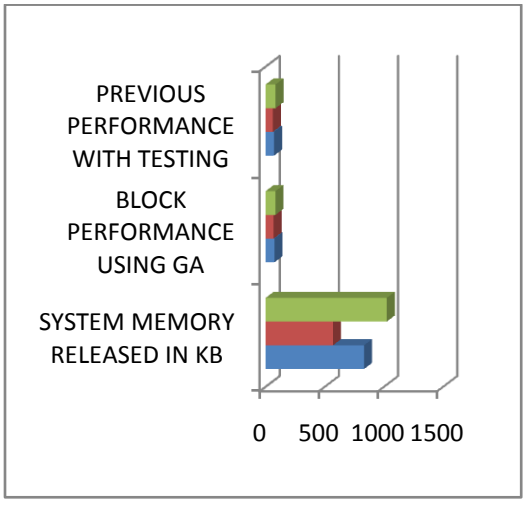


Fig: Banking Software System to be tested

Fig: Result Parameters



The figure represents the overall performance of the system with GA and without GA. The block performance means the overall performance of the system in each and every iteration. We have taken three iterations for testing and it has been found that the overall performance of the system is five to six percent better than without any modification.

VII. CONCLUSION AND FUTURE SCOPE

The result part shows that the release in occupied memory varies when we perform the analysis with GA .If we talk about the block performance, the results shows good improvement of one to eight percent with our testing approach.In future as the software complexities are increasing day by day, if the performance analysis can be made on the basis of threads working over the system, there is always a chance of improvement.

VIII. REFERENCES

[1] Roger Pressman (1997) *"Software Engineering" A Practition-er's Approach 5th Edition, McGraw Hill*.

[2] Beizer B. (1990) *Software Testing Techniques 2nd Edition, International Thomson Computer Press*.

[3] Xanthakis S., Ellis C., Skourlas C., Le Gall A., Kastiskas S., Karapoulios K. (1992) *In 5th International Conference on Soft-ware Engineering and its Applications*, 625-636.

[4] Alan C. Schultz, John J. Grefenstette, aid Kenneth A. De Jong (1993) IEEE- *Test and Evaluation by Genetic Algorithms, Digital Object Identifier*, 8(5).

[5] Pei M., Goodman E.D., Gao Z. and Zhong K. (1994) *Automat-ed Software Test Data Generation Using A Genetic Algorithm*.

[6] Hunt J. (1995) *Testing Control Software using a Genetic Algo-rithm", Working Paper, University of Wales, UK*.

[7] Roper M., Maclean I., Brooks A., Miller J. and Wood M. (1995) *Genetic Algorithms and the Automatic Generation of Test Data*.

[8] Alander J.T., Mantere T. and Turunen P. (1997) *Genetic Algo-rithm Based Software Testing*.

[9] Jones B., Sthamer H. and Eyres D. (1996) *Software Engineer-ing Journal* 11(5), 299-306.

[10] Michael C.C., McGraw G.E., Schatz M.A. and Walton C.C. (1997) *Genetic Algorithms for Dynamic Test Data Generation", Technical report, Reliable Software Technologies, Sterling*.

[11] Tracey N.J., Clark J., Mander K. and McDermid J. (1998) 13*th IEEE Conference in Automated Software Engineering, Hawaii*.

[12] Pargas R.P., Harrold M.J. and Peck R.R. (1999) *Journal of Software Testing, Verification and Reliability*.

[13] Lin J.C. and Yeh P.L. (2000) *9th Asian Test Symposium*.

[14] Bueno P.M.S. and Jino M. (2000) *Fifteenth IEEE International Conference on Automated Software Engineering*, 209-218.

[15] Wegener J., Buhr K. and Pohlheim H. (2002) *Genetic and Evolutionary Computation Conference*.

[16] Daz E., Tuya J. and Blanco R. (2003) 18*th IEEE International Conference on Automated Software Engineering*, 310-313.

[17] Berndt D.J., Fisher J., Johnson L., Pinglikar J. and Watkins A. (2003) *Thirty-Sixth Hawai`i International Conference on System Sciences*.

[18] Tonella P. (2004) *ACM SIGSOFT international symposium on Software testing and analysis*, 119-128.

[19] Berndt D.J., Watkins A. (2005) 38*th Annual Hawaii Interna-tional Conference on System Sciences, Track* 9.

[20] James Miller, Marek Reformat and Howard Zhang (2006) *Science Direct, Information and Software Technology*, 48, 586 - 605.

[21] AbdelhamidBouchachia (2007) *IEEE, Seventh International Conference on Hybrid Intelligent Systems*.

[22] Chen Yong and Zhong Yong (2008) *Fourth International Con-ference on Natural Computation*.