Volume 3, Issue 12, December 2013 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

# A Survey of the use of Genetic Algorithms in Structural Testing

**Michael Scott Brown, Fritzner Neptune, Gina L. Bohl, Megan A. Cifuentes, Jamar Young, Yves G. Tsobgni**
**Anup Garg, Tommy E. Bennett, Nathan B. Hall, Rosalind Winder-Thomase, Sandeep Bhanji, Alemayehu Sissay**
*The Graduate School, University of Maryland University College*
*USA*

***Abstract— This is a survey paper of the user of Genetic Algorithms in Structural Testing. Structural testing like statement coverage or branch coverage can be automated through a variety of algorithms. Genetic Algorithms can be very useful in discovering test cases and test data to implement structural testing. A variety of algorithms are described in different categories of structural testing.***

***Keywords— Software testing, Structural testing, Genetic Algorithms, Statement coverage, Branch coverage***

## I. INTRODUCTION

Structural testing is simply deriving tests based upon the structure of a program. While this certainly is not the only method to derive tests, structural testing is very important for many reasons. Structural testing often provides clear metrics as to how much of the program is being tested for a set of test scripts. Other types of testing, like deriving test scripts from requirements, may inadvertently miss testing areas of the software. Structural testing is often part of any testing strategy. It can be difficult to derive sets of inputs to cover all areas of the structure of a program. Ideally, we would like to minimize the number of test scripts while still meeting some level of structural coverage of the program. Test scripts take time to run so there is a clear advantage to finding ways to reduce the number of test scripts if it can be done without reducing testing coverage. This problem ultimately is an optimization problem. There are many algorithms that perform optimization. One of them is Genetic Algorithms (GA), which can be very useful in structural testing. This is a survey paper of the use of GAs in structural testing. There are far too many research papers on structural testing methods that use GAs to include in a single survey paper. Papers were selected based upon their date and to provide a variety of papers in different areas. But the papers selected will provide a good overview of the types of research being conducted today.

## II. GENETIC ALGORITHMS

GAs are a search technique that models biological selection. In biology a species will cope with environmental factors. Individuals that cope better with the environment are more likely to survive and reproduce. During reproduction they pass their genetic traits to their offspring. The next generation is better at coping with the environment. Early research in Artificial Intelligence took this concept and created a search algorithm based upon this natural selection.

In a GA domain values are modelled as a string of characters. This is normally binary and modelled as a series of 0's and 1's. This is also a *fitness function*, which models the problem that the GA is attempting to solve. Each individual has a fitness value, which is computed through the fitness function. The fitness value shows how well the individual is at solving the problem.

A GA normally begins with randomly generating individuals. This set of individuals is called the first generation. Then it goes through a process to produce the next generation. The first step of this process is to select pairs of individuals to reproduce. Individuals with higher fitness values are more likely to get selected for reproduction. This step is called *selection* and an individual can be a member of multiple selection pairs. Each selection pair is used to create two new individuals. This is done by randomly selecting positions within the individuals to break them apart. The new individuals are the combination of the alternating parts of the two original individuals. This step is referred to as *cross-over*. The final step is mutation. Mutation is based upon a random event and when it occurs a value of the string of an individual is changed. This allows the search to look at new areas of the domain. This new set of individuals is the next generation.

GA theory tells us that over time the generations will converge to the optimal value of the domain. This iterative process searches the domain and improves upon the previous generations. Selection and cross-over are convergent forces that narrow the solution space. Mutation is an exploratory force that attempts to investigate new areas of the domain. Over time the convergent forces overtake the exploratory forces and the generations converge on the optimal value.

## III. STRUCTURAL TESTING

There are many different types of structural testing and there may not be agreement of the different categories of structural testing. This paper categorizes them into six categories: statement testing, branch testing, condition testing, path testing and procedure call testing.

 

Statement testing is identifying and running a set of test cases that attempt to execute each statement of a program. The statement coverage is the ratio of the number of statements executed by the test scripts divided by the total number of statements. Branch, condition and path testing are identical except they measure the coverage of branches, conditions and paths respectively. Procedure call testing is identifying and running a set of test cases that execute each time a procedure is called.

Some of the algorithms presented in this paper are for a certain type of structural testing. However, many of the papers suggest that the algorithms could easily be adapted to accommodate other types of structural testing.

## IV. GENETIC ALGORITHMS USED IN STRUCTURAL TESTING

### *A. Nachiyappan, Vimaladevi and SelvaLakshmi*

The Nachiyappan, Vimaladevi and SelvaLakshmi [1] algorithm attempts to address the problem of test suite reduction for regression testing. Regression testing can be very time consuming and by selecting a subset of test cases that provides a certain level of coverage can greatly cut the time needed to test. The researchers developed a mathematical model, to model the problem of generating a test suite with the largest coverage and at a reduced run time.

To accomplish test-suite reduction the algorithm models the problem as an optimization problem. Subsets of the known tests cases are modelled as Individuals. The fitness function is $f(t) = coverage(t) / time(t)$. In this equation $t$ is the individual, $coverage(t)$ is the coverage of the program that the subset covers and $time(t)$ the time that it takes to execute the test-suite. The function $coverage(t)$ could mean any type of coverage statement coverage, branch coverage, etc. Unlike most GAs this algorithm will perform cross-over or mutation, but not both. This is based upon a random number and a probability of cross-over parameter.

The GA has the following parameters: Test-suite, Coverage, Run-time, Mutation Rate and Crossover Probability. The result of the GA is a reduced test-suite. This represents the test-suites that showed a significant reduction from the original test-suite. This approach allowed for the creation of genes that would get missed in a straightforward crossover methodology. The algorithm is a hybrid method of both mutation and crossover of individuals. This ensured that unlikely options were at least explored, which one could speculate would increase the chances for a more optimized test suite that wouldn't be created during traditional crossover alone.

The research conducted revealed that better test suites can be generated using GA by both limiting and focusing the testing that can produce the best results. GA generated significantly smaller test suites that required less time to run, thus saving money and increasing quality. Test suites generated by people are limited by design, whereas GA will allow for the evolution of possibilities and combinations that would not have been considered by a person. While inclusion is the focus of finding the best test suite, GA can also perform exclusion from the test suite as well. The exclusion can be based on certain factors such as test cases that have to narrow of coverage that yields no real benefit or the test coverage can be found in another test case. The elimination of test cases would be difficult for a person to perform with any speed and precision. Because of GA's ability to create test suite optimally, it can be assumed that the rate of successful fault detection would be better than that of a traditionally created test suite as well.

### *B. Khor and Gorgono's Genet*

Genet utilizes a combination of genetic algorithms (GA) and Formal Concept Analysis (FCA) to create an Automatic Test Generator (ATG) that ensures branch coverage [2]. FCA is a mathematical theory, which utilizes a combination of objects and attributes [3]. FCA discovers the hierarchical relationships between a set of objects and a set of attributes. In the example of software testing the set of objects are test cases and the set of attributes are branches within the program.

Genet uses an iterative process to identify branches within a program [2]. Once all test data has been used, a calculation is done to see if branches still exist that have not been uncovered. If this calculation returns a result that is below the stopping point for testing then formal concept analysis is used [2].

The program is run using the test data either from a file provided by the user or randomly generated and a record is kept of the execution. The loop of the algorithm contains a number of steps. First each individual in the current generation is executed in the target program the each branch that is taken is recorded. If all branches are reached or the GA has hit the maximum number of generations the GA terminates. Next the GA uses FCA to organize the individuals into hierarchical groups called *concepts*. A concept is a subset of tests and branches in which each test explore all of the branches. Concepts are ranked and each individual is ranked according to how many times it appears in the winner concept and its super concepts in the hierarchy. Individuals are sorted by this ranking and based upon a parameter the lower ranking ones are removed. Of the remaining individuals the cross-over and mutation operations are applied to produce a new generation of individuals. This loop continues until the termination condition is met and the final generation is the test set.

By combining genetic algorithms and formal concept analysis methods, genet is able to uncover more relationships then previous ATG methods, and therefore results in more thorough testing.

### *C. Harman, Lakhotia and McMinn*

The Harman, Lakotia and McMinn [4] algorithm is another ATG. However, it attempts to optimization multiple objectives like branch coverage and memory usage. This is referred to as the multi-objective problem (MOP). The MOP is explained as the challenging effort of generating test data to simultaneously meet several test objectives to lower costs and time while producing high quality results. The authors suggest a multi-objective approach is most reasonable while testing various objectives because the current approaches do not satisfy additional external testing goals that may arise. Genetic algorithms are presented that incorporate the fitness function, which seeks values that are closest to a specified objective for highest optimization. The five case studies conducted in this article seek the optimal solutions (best fit

 

value) with the algorithms, for two objective functions, the highest branch coverage and most dynamic memory consumption, in both real and simulated programs to test extreme scenarios. Therefore, the article details the results of the case studies by using genetic algorithms to determine which approach is best for testing multiple objectives. The optimization in the case studies was found using two GAs: Pareto GA and Weighted GA.

Pareto GAs are ideal for finding optimal values in cases where there are multiple objectives that in some way compete against each other. Harman, Lakotia and McMinn [4] contend that ATG often experience this situation. In their example of test data that covers the most number of branches while minimizing memory is such a case. Pareto GA finds various optimal solutions by creating several front lines with non-dominating solutions. These set of optimal values are referred to as the *Pareto Front.*

Pareto GA fitness function computes fitness values by ranking individuals within a population according to distance between objectives and neighboring value in a front. Pareto GA rewards diverse and spread out individuals within each population. The fitness function helps quantify this reward by combining the distance from the objective and the nearest individual for each value within a population to rank the individuals properly. In other words, individuals that are in less crowded regions of a front are rewarded while those in crowded regions are not. In addition, the lowest ranked individual of one front has a better fitness value than the highest ranked individual in the following front. Thus, the individual with the greater distance from its neighbor will rank higher within the population, and have a greater chance of optimal selection.

A weighted GA calculates a single best solution and uses linear rank among populations rather than front lines that have several rankings. This algorithm is best utilized when objectives are prioritized in a significant manner. The individuals are modeled by how the objectives are prioritized numerically. For instance, highest branch coverage may be a higher priority than dynamic memory consumption, which is reflected in the fitness function by a higher weight, or percentage.

Weighted GA fitness function considers the branch coverage objective as minimization, which inverses a normal optimal solution that usually finds the maximum value. For that reason, the weighted GA must use the inverse of the normalized number of bytes allocated in the memory consumption objective with the distance measures. Thus, an individual that is the closest to zero is considered the optimal value for an objective.

After conducting the case studies, the authors determined that a hybrid multi-objective approach would work best that emphasizes GAs. However, there are some disadvantages to these algorithms. Pareto GA involves higher computational cost while weighted GA requires an efficient set of objective weights. If objectives are not properly weighted, the results are negatively affected. The authors also discovered that the Pareto GA does not produce a front–line and 100% branch coverage is very hard to achieve for programs containing *malloc/calloc/realloc* statements. Furthermore, the authors acknowledge hierarchical GA as a potential technique for larger real world programs and are overlooked in this article's case studies. In conclusion, a hybrid multi-objective approach with GAs is most effective using generated test data in most scenarios.

*D. Ahmed and Hermadi*

GAs are often used as ATG for path coverage. Many of them only search for single target paths. The Ahmed and Hermdai [5] algorithm attempts to address this problem. The algorithm designed is a GA-based test data generator that synthesizes multiple path test data in one run. This research showed that the generator was more efficient and effective than others.

The authors introduce a term called *Building Blocks*. This is an equation used in the GAs fitness function. It sums up the difference between the predicate values and the traversed nodes between the target path and the path of the individual in the GA. For building blocks, the authors compared traversed paths and target paths in terms of distance D and violation VA. D is the difference between the two paths in terms of predicate values for the unmatched node-branches. V is the number of unmatched nodes between the target path and the path of the individual. The building block is a good model for determining the difference between the path created by the potential text data of the individual of the GA and the target path. Once this difference can be modeled it becomes a minimization problem, which is something that GAs are very good at solving.

This intermediate fitness function can be more formally defined by the following equations:

$$IF_{ij} = D_{ij} + V_{ij}, \quad (1)$$

$$D_{ij} = \sum_{k=1}^{n} D_{ijk}. \quad (2)$$

$$V_{ij} = \sum_{k=1}^{n} V_{ijk} \quad (3)$$

Where, *i* is index of target path; *j* is index of individual; *k* is index of node in both target path *I* and traversed path *j*; and *IF* intermediate fitness that looks at the fitness of an individual with respect to one target path. *IF* is considered as a building block for the overall individual fitness where the fitness with respect to all target paths is considered. The predicate value is calculated using Korel's distance function [6]. The distance equals to zero if the node-branch of both the target path and the traversed path are matching.

 

The distance *D* was normalized by either all target paths along or with other individuals. The violation *V* was normalized by the length of its target path. Path traversal method measured closeness between the path exercised by an individual and a target path. In GA, a chromosome has 2 types of fitness values; its own fitness value, and the value influenced by the target and/or other neighboring chromosomes competing to cover similar targets. Accordingly distance normalization schemes were implemented to normalize the two values. Weighting were considered to differentiate between the contribution of distance *D* and violation *V* to the overall fitness. An adjustment value was assigned by the designer of the test generator to refine the fitness function. The individual having highest overall fitness, for a target path, was rewarded because it has a better chance of survival to the next generation. Overall fitness value was obtained by summing up and normalizing all the *IF* values of each chromosome. The term 'Normalization' used in this paragraph (say for *IF*) means dividing the *IF* values of each chromosome by the sum of all *IF* values, so that the sum of all resulting *IF* values equals 1.

Their experimental results showed that their candidate fitness function was effective and efficient in handling the required feasible target paths, regardless of existence of infeasible paths, the path length, and the compound predicates complexity. The existence of infeasible path (if any) rather helped in exploring the search space. In general, predicate-wise candidates were slightly more effective than path-wise ones, while path-wise candidates were found to be more efficient than predicate-wise ones. The fitness functions that were utilizing neighborhood influence were found to be more effective but not more efficient as they took more computation time. Candidates applying rewarding scheme and/or violation-based weighting were more exploitive than their counterparts. Based on the above results, the authors reported their test data generator as more effective and more efficient than existing generators as it allows covering multiple target paths with less number of test data examined.

*E. Mansour and Salame*

Mansour and Salame [7] present two algorithms for automating testing at the unit level following white box testing methods. Only one of the algorithms is a GA, so that will be the focus of this section. Each individual in this population corresponds to a test case along with the input values associated with the test case.

The reproductive process is described by the authors to be based on natural selection processes that copy the best performing chromosomes to the next generation, fitness level ranking and random chromosome selection. [7] The ranking scheme is based on a numerical value where the higher rankings are duplicated, the middle values are randomly selected for duplication and the lower performers are removed from the evolution. As duplication occurs mutations are introduced into the population to ensure diversity and the possibility of improvement. Mansour and Salame [7] define convergence as no improvement to the monitored best-case candidate solution.

The authors were able to demonstrate the capability of the presented algorithms. Their empirical evaluation results showed that the genetic and simulated annealing algorithms were able to cover more paths than the Korel's algorithm. The simulated annealing algorithm is more effective than the others, however, the genetic algorithm is faster. A noteworthy observation from the authors was that the Korel's algorithm was the fastest algorithm when it actually was able to perform the required evaluation.

*F. Mohapatra, Bhuyan and Mohapatra*

The Mohapatra, Bhuyan and Mohapatra [8] algorithm aims to automate and optimize test case generation and path testing. It uses a GA for optimization and expands its scope to include sampling statistics methodology to assist in automating and finding the optimal test suite. In this algorithm a GA is used in combination with random sampling to generate optimized test case suites. First, the initial test cases population are generated from the Test Harness and provided to the GA to initiate the optimization process. Then the GA is executed to search for all testing paths based on a fitness function that defines the structure of the solution. This optimization process results in producing a new population of test cases, where the flow control graph – a graphic representation of the module or the Unit Under Test (UUT) – is presented to the next step of the process. To assure the completeness and effectiveness of the solution in its objective, a sampling methodology complements the GA. A number of sampling methods are presented including random, binomial and stratified sampling. The sampling method is used to derive a set of test cases that equal the Cyclomatic complexity from the flow control graph to obtain the optimal test suite.

Mohapatra, Bhuyan and Mohapatra [8]is a short research paper and it is a little unclear what individuals in the GA are modeling. But combining the GA with the Harness Pattern is a novel approach to testing. The research is also novel in the approach of determining the test case size by using the Cyclomatic complexity.

*G. Bueno and Juno*

POKE-TOOL [9] is a set of tools used to support the data flow based Potential Uses criteria. The tool can also be used to generate test data for other data or control-flow criteria by selecting appropriate sets of paths. The testing criterion requires the execution of paths from points where variables are assigned, values (definitions), to points in the program where the definitions (values) can be used. The tool can also be used to generate test data for other data or control flow criteria by the selection of appropriate sets of paths.

The symbolic execution and the dynamic technique are approaches for the generation of test data in structural and fault-based testing. The symbolic execution is a static representation of the execution conditions, for the program's input variables of a given path and a search is created to find solutions that satisfy the conditions and cause the path's execution. The static nature of this approach places restrictions on the treatment of loops, arrays and pointers.

The dynamic nature of the testing, along with the carefully defined instrumentation model, makes it possible for the technique to handle arrays, strings, pointers, loops and complex arithmetic expressions. In the dynamic technique, actual values are assigned to input variables and the program's execution flow is monitored. If the program flow deviates from

 

the intended flow, searches or numerical methods are used in an attempt to find input values, which will set the desired program flow. [9]

The structure of a program *P* can be represented by a directed graph called Control Flow Graph *(CFG) G = (N, E, s, e),* where *N* is a set of nodes, *E* a set of edges, ***s*** the unique entry node, and ***e*** the unique exit node. A node is a set of statements that are always executed together. A path is a node sequence and the number of nodes on a path is its length. An edge ($n_i$, $n_j$) is called a branch and corresponds to a possible control transfer between the nodes $n_i$ and $n_j$ .

The GA is used to explore the program's input domain and to search for an input data set that executes the desired path. The GA's ability to handle complex restrictions and to accurately define the fitness functions has resulted in the successful generation of test data. The following fitness function, shown in equation 4, is used to evaluate each input data set. *NC* represents the coincident nodes between the executed path and the desired path. *EP* is the predicate function absolute value associated to the branch where there is a deviation from the desired path. *EP* also represents the error that causes the executed path to deviation. *MEP* is the predicate function's maximum value, of the input data sets, in the current population, that execute the same nodes of the desired path. [9]

$$Ft = NC - \frac{EP}{MEP} \qquad (4)$$

The Fitness Function is executed in three steps. In the first step, *NC* and *EP* values are computed for each data set. In the next step, *MEP* is computed for each predicate along the desire path and finally, the *Ft* is computed for each input data set.

The greater the number of correctly executed nodes, the closer the input data set to the desired solution, defines the fitness function. The larger the fitness value, the better the input data set. Fitness values are used to drive the search needed to reach the desired path's nodes and to solve the desired paths' predicates.

The predicate function values (*EP*) are computed using real values involved in predicates, obtained by the analysis of dynamic data flow information generated from program executions. The fine-grained distinction among the solutions determines the technique's high success rate in test data generation.

Each candidate solution individual represents an input data set, which is a concatenation of values assigned to the program's input variables. Using these operators, the algorithm creates the next generation from the current population of solutions, by combining them and by inducing changes. The iterative combination and selection of solutions cause a continual evolution of the population towards the problem solution.

The automated test data generation program for program paths using GAs is instrumented by selectively inserting statements to provide information on the execution needed to drive the data generation. The instrumented program contains file-writing statements needed to get dynamic information on the control and data flows during the program's execution. The statements are inserted just after decision statements, to save the real values involved in expressions or variable values, relational and logical operators, and the program node executed. This information is then used to compute the path and the predicate function. Incorporating advances from GA applications, on constraint optimization problems, along with the investigation of specialized genetic operators for test data generation, is some of the continued work being conducted. The efficient solution of linear predicates, the adaptation of the technique for goal oriented and error-based test data generation and the integration testing and conducing empirical evaluations with more complex programs is also part of the ongoing and future work being done to perfect the automatic test data generation for program paths using genetic algorithms. [9]

*H. Rajapph, Biradar and Panda*

Rajappa, Biradar and Panda [10] created a new method for generating efficient test cases using a graph theory based GA. The graphs represent states of the program and the GA attempts to find at path through the states for use in testing. In this algorithm individuals model graphs that represent state and state transition within a program. An independent set in graph theory is the subset of vertices such that no two vertices share an edge in the graph. A clique in graph theory is the independent set of the complement of the graph. The fitness of an individual is the number of cliques of size 5 within the graph plus the number of independent sets of the size 5. This allows the GA to locate testing paths through the program.

The graph theory model will be necessary for completion of the entire graphs to allow illustration of the conditions of the arrangement [10]. This will be facilitated by first creating straight graph including the transitional states. To generate the nodes for the child, the total graph is covered for instance once put into place total traversal. Relating to the scope of crossover, a new population will be found. This practice would issue the tester with a permanent state of the crucial mapping regarding the system in the operating state. This will result to state changeover and load investigation will require the process of testing in definite test outcomes in addition to the expected condition.

In conclusion the evaluation of fitness should be considered to evaluate the most applicable graph to the software testing process. With the aim of performing the tournament, a correct approach of categorizing the vital individuals completely should be set. Therefore, the state most advantageous system should be adopted which follows that, the likelihood of unproven paths relating to system actions must be reduced, and also the analysis coverage in the end grows.

*I. Arcuri*

Arcuri [11] investigates another interesting problem with automated testing. The problem of creating a test suite that fulfils some type of structural coverage is more complex when the software components have state. In Object-Oriented Programming objects can have state. This state has an effect on structural testing coverage. Each test case will have to put the object in a desired state through method calls before the desired method to test is called. Each test case will have a sequence of method calls.

 

This creates an interesting testing problem. Ideally we would like to obtain some level of coverage, like branch or path coverage. However we would like to minimize the sequence of method calls. One of the algorithms that Arcuir [11] used to develop test cases is a GA.

Software Testing and sequence length have a fascinating relationship together. The way a sequence's length within the internal state of the software plays a role in which a search algorithm is selected. Different search algorithms are used to find a sequence length. The search algorithms are Random Search (RS), Hill Climbing (HC), Evolutionary Algorithm (EA) and GA. Each technique has a way of operating for RS a constant scattering (ratio) is used, for HC the search method analysis the surrounding solutions to find if a better fitness value. EA runs one evolutionary algorithm, which includes mutation of the parent. The GA uses elitism and a ranking based system for selection of the most optima fitness function. The focus is how each search algorithm performed when it came to inserting and removing objects.

The deciding factor when it comes down to selecting a search algorithm to perform the search length. The noticeable results from the analysis are that each technique performed better the greater the length resulting in more optimal solutions. The overall goal is to find the global optimal for all search algorithms. The search algorithms are tested in a case study in various containers (TreeMap, Hashtable, Vector, LinkedList, BinTree and BinomialHeap). Each search algorithm is tested in but the TreeMap is focused in this paper. The analysis of each method shows the increased number of runs seem to produce a more global optima. When running an actual search the main criterion is based on the average number of methods executed.

Each search algorithm produces certain results based on the length; an example is for a small segment length the GA could outperform the EA because GA wants to quickly find a solution, thus keeping the search time to a minimum. Once the best solution is found the algorithm stops running. Under some circumstances running a short sequence (length) could produce some results but when it comes to trying to shorten the length short sequences become unfeasible.

When used the container classes in sequence with longer test sequences yielded better results. But in reality, the container classes are only a small precent of the Object Oriented software, so conditions are applied to create the best results. GAs work by finding the fit solution and creating a duplicate copy. Then the GA performs mutation. If the mutation has a higher elitism rate then the previous generation it is passed on if not it is removed. The GA results in this case are modeled by running them on different segment lengths. By using different lengths the data can be analysed to find at what length and number of methods calls will the GA perform the best. The GA is modelled using the different containers mentioned previously each container has a different configuration which yields different results.

The complexity of the software can play a role in the actual test sequences. In addition, producing real problems that are needed and can be used in a search algorithm to perform these test sequences can result in a more comprehensive understanding of the length of test sequences. First develop an understanding of how different lengths of test sequences can produce different results. Then see how the test sequences length results change by implementing them in real software. Study the results to achieve an understanding how different methods achieve certain result in software testing. The difficulty is taking a broad topic and applying certain techniques to try to predict the outcome. Once more time is spent on sequence length for software testing a clearer understanding how it could affect real software.

*J. Weimer, Nguyen, Le Goues and Forrest*

Weimer, et al. [12] took a completely different approach. Their algorithm actually attempts to fix the defect. It does this through knowing what test case failed and the complete set of test cases that covers this area of code. The complete set of test cases is used to ensure that the bug fix still implements the requirements of the section of code. Another assumption that the authors make is that the defect fix is similar to code in other areas of the program. There is a pattern in other areas of the program that are the solution to this bug. Their example of such a defect is checking for null values.

This algorithm uses Genetic Programming (GP), which is similar to GAs. Variations of the code are created, called variants. Compiling abstract syntax tree of the code and running it on the test cases determine the quality of each variant. The final results are determined by the sum of the positive and negative tests. The GP will work by searching through the space of nearby variants until it finds the known defects and will retain their functionality. Mutation is usually linked to the weighted path. For crossover, only statements along the weighted path will be included in this group. Crossover always takes place between individual from the current population and the original parent program.

There is a need to go through a different type of selection in order to find the right function. When the selection is complete, the right function will be put in place. It will return a number indicating the acceptability of the program. The selection algorithm usually uses this number in order to determine which variants go into the next iteration. The fitness function is also known for encoding software requirements at the test case level. The negative part will be considered for repair and the positive one will encode the functionality that cannot be sacrificed. A high success rate would rely on the luck to find the issue, but a low rate means that circumstances should align in order to find the repair.

One limitation to this approach is that the defect must be reproducible. In non-determinist programs this is not always possible. Running many times the test cases can mitigate this limitation.

A fully automated technique is used for repairing bugs in off the shelf legacy software. Weimer, et al. [12] adopted instead an extended form of genetic programming to evolve program variants. One part of the program is used as a template in order to repair another part. GP relies on a combination of abstract syntax trees with weighted violating paths. Standard tests cases are used to show the fault and to encode required functionality. With this method the repair cost is tremendously minimized which is a good sign in order to generate profit for companies.

*K. Girgis*

The Girgis [13] algorithm is another ATG. But it focuses over def-use path coverage and provides a list of test data along with the def-use paths that the test data covers. When a variable is assigned a value it is called a *def*. When a

 

variable value is used it is called a *use*. By looking at a program in this way we can define def-use pairs. These are the combinations of places were the value of a variable is defined and a place where that value is used. For each def-use pair all of the paths between the *def* and the *use* is a def-use path. The Girgis algorithm attempts to find test data that maximizes def-use paths.

The algorithm [13] models individuals as values for the input variables of the program to be tested. It created individuals of the appropriate size depending upon the precision of the input variables. The algorithm produces a set of test cases. The algorithm also uses an integer vector to record the traversed paths. The fitness of each individual is the number of de-use paths it covers divided by the total number of def-use paths.

To test [13], the results from GA technique and Random Technique (RT), based on the results of 15 programs are compared. From the results it can be concluded that, GA technique has outperformed the RT in 12 out of the 15 programs. In case of the GA techniques, 10 programs required lesser generations or iterations than RT to achieve the same level of results. In one case the RT required 51 generations to cover 58.4% of the paths, while the GA technique required only 15 generations to cover 63.6% of the paths. Further such results were neither unique nor isolated. In several other examples [13], the GA covered more paths with fewer generations as compared to the RT.

Further, the results of two parent selection methods were compared [13]. Based on the results, the proposed RS method was found to be significantly better than the RW method. These results held true in 7 cases and only in one case the RW method was better than the RS method. In other 7 cases the results of both were identical. Based on this, the RS method was found to be superior to the RW method.

## V. SUMMARY AND CONCLUSIONS

Structural testing is a very important technique in testing. Structural testing ensures that all areas of the program are tested. This in combination with other testing techniques can make up a good test plan. This survey paper outlines a small subset of research in structural testing. Specifically it covers algorithms that use GAs to perform structural testing. Even within this area of research the number of algorithms were too many to include in a single paper. This paper provides an overview of different types of algorithms.

One observation that can be made is regarding how GAs are used in these algorithms. Some of the algorithms, like Mohapatra, Bhuyan and Mohapatra [7] are very low level. Individuals are modeled after actual input data to the program or procedure. Other algorithms work at a higher level, like Nachiyappan, Vimaladevi and SelvaLakshmi [1]; Khor and Grogono [4]; Harman, Lakotia and McMinn [4]; Ahmed and Hermdai [5].These algorithms assume a test suite has already been created and attempt to find a subset of test cases that meet some condition. Mansour and Salame [7] takes a hybrid approach. Here an individual models a test case along with the input data to the test case. Still other research in the area attempts to actually fix the defect, Weimer, et al. [12]. GAs are very useful for many types of structural testing.

GAs are very useful search algorithms. They are used in many areas of research. This paper illustrates their usefulness in the area of structural testing. GAs can develop procedure input data and perform test case reduction, along with many other uses.

## REFERENCES

[1] S. Nachiyappan, A. Vimaladevi and C. B. SelvaLakshmi, "An Evolutionary Algorithm for Regression Test Suite Reduction", *Proceedings of the International Conference on Communication and Computational Intelligence*, pp. 503-508, 2010.

[2] S. Khor and P. Grogono, "Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically", *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pp. 346-349, 2004.

[3] L. Wang, X. Liu and J. Cao, "A new algebraic structure for formal concept analysis", *Information Sciences*, vol. *180*(24), pp. 4865-4876, 2010.

[4] M. Harman, K. Lakhotia and P. McMinn, "A Multi-Objective Approach to Search-Based Test Data Generation", In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pp. 1098-1105, 2007.

[5] M. A. Ahmed, I. Hermadi, "GA-based Multiple Paths Test Data Generator", *Computers & Operations Research*, vol. 35(10), pp. 3107-3124, 2008.

[6] B. Korel, "Automated Software Test Data Generation', *IEEE Transactions on Software Engineering*, vol. 16(8), pp. 870-879, 1990.

[7] N. Mansour and M. Salame, "Data generation for path testing", *Software Quality Journal*, vol. 12(2), pp. 121-136, 2004.

[8] B. Mohapatra, P. Bhuyan and D. P. Mohapatra, (2009). "Automated Test Case Generation and its Optimization for Path Testing Using Genetic Algorithm and Sampling", *Information Engineering, 2009. WASE International Conference*, pp. 643-646, 2009.

[9] P. M. Bueno and M. Juno, "Automatic test data generation for program paths using genetic algorithms", *International Journal of Software Engineering and Knowledge Engineering*, vol. 12 (6), pp. 691-709, 2002.

[10] V. Rajappa, A. Biradar and S. Panda, "Efficient Software Test Case Generation Using Genetic Algorithm Based Graph Theory", *First International Conference on Emerging Trends in engineering and Technology*, pp. 298-303, 2008.

[11] A. Arcuri, "Longer is Better: On the Role of Test Sequence Length in Software Testing", *Third International Conference on Software Testing, Verification and Validation*, Paris, France, pp. 469-478, 2010.

 

[12] W. Weimer, T. V. Nguyen, C. Le Goues and S. Forrest, (2009). "Automatically finding patches using genetic programming", *International Conference on Software Engineering*, Vancouver, Canada, pp. 364-374, 2009.

[13] M. R. Girgis, "Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm", *Journal of Universal Computer Science*, vol. 11(6), pp. 898-915, 2005.

 