Journal of Computer Science 4 (12): 1012-1019, 2008
ISSN 1549-3636

A Complete Automation of Unit Testing for JavaScript Programs

Mohammad Alshraideh
Department of Computer Science, University of Jordan, Amman 11942 Jordan

Abstract: Problem statement: Program testing is expensive and labor intensive, often consuming more than half of the total development costs, and yet it is frequently not done well and the results are not always satisfactory. The **objective** of this paper is to present an automatic test data generation tool that aims to completely automate unit testing of JavaScript functions. **The methodology**: In order to use the proposed tool, the tester annotates the files that contain the class to be tested. Moreover, the tester must specify the test data coverage criterion to be used, either branch coverage or mutation analysis. However, the tool is then integrated into the JavaScript compiler and test generation is invoked by a command line option. Also, the code to be tested is parsed into an abstract syntax tree from which the test tool generates a program dependency graph for the function under test. However, if mutation analysis coverage is required, the abstract syntax tree for a meta-mutant program is also generated. To provide guidance for the test data search, the function under test instrumented in accordance with the coverage criterion. Branch predicate expressions are always instrumented, in the case of mutation coverage, mutated statements are also instrumented. Compilation then continues from the modified abstract syntax tree to generate instrumented executables that were loaded into the test data search module. **Results**: The experiment done in our study by using the proposed tool for branch coverage shows that the most effective result for string equality was obtained using the edit distance fitness function, while no significant difference was found in the fitness function for string ordering. Through exhaustive mulation coverage 8% are found to be equivalent. **Conclusion**: By having a complete automation it reduces the cost of software testing dramatically and also facilitates continuous testing. It is reported that at least 50% of the total software development costs is due to testing, and 10–15% of development time is wasted due to frequent stops for regression testing. Automation will also help get rid of cognitive biases that have been found in human testers. **Acknowledgment:** The researcher would like to express their gratitude to the anonymous referees for their valuable and helpful comments and suggestions in improving the study.

Key words: Software testing, white box, black box, genetic algorithms, mutation testing.

INTRODUCTION

Software testing is as old as the hills in the history of digital computers. The testing of software is an important means of assessing the software to determine its quality. Since testing typically consumes 40-50% of development efforts and consumes more effort for systems that require higher levels of reliability, it is a significant part of the software engineering. With the development of Fourth Generation Languages (4GL), which speeds up the implementation process, the proportion of time devoted to testing increased. As the amount of maintenance and upgrade of existing systems grow, significant amount of testing will also be needed to verify systems after changes are made[20]. Despite advances in formal methods and verification techniques, a system still needs to be tested before it is used. Testing remains the truly effective means to assure the quality of a software system of non-trivial complexity, as well as one of the most intricate and least understood areas in software engineering [21]. Testing, an important research area within computer science is likely to become even more important in the future.

This retrospective on a fifty-year of software testing technique research examines the maturation of the software testing technique research by tracing the major research results that have contributed to the growth of this area. It also assesses the change of research paradigms over time by tracing the types of research questions and strategies used at various stages. So, the sooner in the development process that a fault is found, the less expensive it is to correct. Unit testing, which occurs at the start of the testing phase has the potential to be a very cost effective form of testing. In

practice, however, unit test cases are invariably constructed manually by a tester who may need to spend significant time analyzing the program under test. Consequently, there is much interest in the prospect of generating unit test data automatically or with assistance. The test data generation tool described in this study attempts to generate test data for the unit testing of JavaScript (also known as JScript or EcmaScript). The code to be tested must reside in a file. To use the tool, the tester must annotate the file to specify the class and the function under test. For each function, the tester must specify the input domain and the test data coverage criterion to be used, only two are available either branch coverage or mutation analysis, DeMillo *et al.*[1] and Hamlet[2], which requires that test data demonstrate the absence of a specified set of faults. In practice, mutation analysis subsumes branch coverage. In addition, some optional annotations may specify the search strategy to be used and other search parameters. The tool is integrated into the JavaScript compiler and test generation is invoked by a command line option.

At the core of the test data generation tool is the test data search module. This module uses a dynamic, search-based approach to software test generation [3-8]. This approach requires that the program under test be instrumented and executed to assess candidate test cases. The program analysis module is responsible for analyzing the program under test so that it may be suitably instrumented.

Background and related works: There are several types of tools in order to facilitate the software testing process and they have different functionalities. Among these functionalities we can find the following ones: to automate the path achieved in source code by a test case, to automate the execution of software tests[22] and to automate the generation of test cases by means of the instrumentation of the source code under test[18,22]. This instrumentation can be in automatic way or by hand.

The program-based approach, such as statement testing, branch testing, condition testing and path testing, generates test data by analyzing the source program to be tested[19]. This approach is practical and supported by several commercial tools; however, it requires separate test oracle code to be written.

Automatic tool description: The several tool has several modules:

- Parser: It generates the control flow graph of the source code under test
- Instrumenter: It generates the instrumented source code
- Test cases generator: It generates the test cases, using the instrumented source code and the control flow graph

The scheme of our tool appears in Fig. 1.

A parser has been developed that generates files with the control flow graph, data flow graph, data dependency graph and control dependency graph from the source code of the program that is going to be tested. Each graph node stores important information that is used in the testing process. The instrumenter then reads the source file and instruments the program under test using the control flow graph.

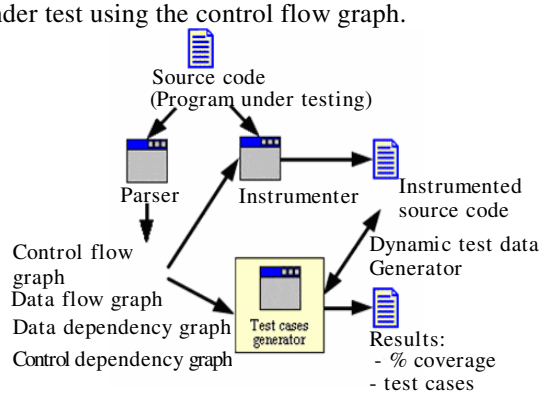

Fig. 1: Automatic tool scheme

Finally, the test case generator is executed from the instrumented source code and its complexity graph: in each iteration it generates test cases for the program under test and executes it with them to store their behaviour. The generator finishes when it obtains a desired branch coverage percentage or reaches the maximum number of attempts allowed.

Input domain specification: The following is an example of an annotated class file. The tester is responsible for writing the class initialiser and the function ExecFUT. This function allows the tester to specify how the Function Under Test, FUT, will be called on an instance of CUT. The class CUT has an instance variable array ***a*** that must first be initialised by calling Init. This function stores multiples of 5 in successive elements. Then any of the function members Inc, Dec or Swap may be called before FUT is called last. The function Inc increments a given element in the array ***a***, Dec decrements a given element and Swap exchanges ***a*** given element with the first element. A sequence of calls beginning with Init and ending with FUT is a test case.

The tester must write a function, ExecFUT that implements a valid sequence of function calls. An

object array is provided into which values of the in-built types may be placed. These values are used to select functions to call and to provide the arguments for them. In the example below, each call within the test case is coded as a segment of an integer array as defined in the class initialiser. The first 18 elements contain 6 segments of 3 elements. The first element of any 3 determines which function will be called, as implemented by the switch statement. The second element is the argument to Inc or Dec; the third element is the argument to Swap. A single value may encode the argument to both Inc and Dec but in general a different value is required for each function as is the case for the argument to Swap:

```
public class CUT {
static CUT {
var domfun: Doma1inInt32 = new DomainInt32 ([1, 4, 2], [[0, 0], [1, 2], [3, 3]]);
var domincdec: DomainInt32 = new DomainInt32 ([1], [[0, 5]]);
var domswap: DomainInt32 = new DomainInt32 ([1], [[1, 5]]);
GA.Init ([domfun, domincdec, domswap,
domfun, domincdec, domswap,
//4 MORE ROWS AS ABOVE
domswap]);
  }
var a : int[] = new int[6];
function Init() {
    var i : int = 0;
    for (i = 0; i < 6; i++) {
     a[i] = i * 5;}
}
function Inc(n : int) {
    a[n] = a[n] + 1;
 }
function Dec(n : int) {
     a[n] = a[n] - 1;
}
function Swap(n : int) {
   var temp : int = a[n];
   a[n] = a[0];
   a[0] = temp;
 }
public function FUT(n : int) : int {
   if (a[0] == a[n]) {
    return 1}
    return 0;
}
function ExecFUT(gen : Genotype) {
    Init();
    var i : int;
    i = 0;
    while (i < 18) {
    switch (gen.chron[i]) {
    case 0:
        break;
     case 1:
        Inc(int(gen.chron[i + 1]));
        break;
      case 2:
        Dec(int(gen.chron[i + 1]));
         break;
       case 3:
         Swap(int(gen.chron[i + 2]));
          break;
         }
       i = i + 3;
 }
 Subject(int(gen.chron[18]));
   }
 }
```

The tester also specifies the domains of the arguments to each function by type and value constraint. JavaScript is not strongly typed. In the absence of type information, function arguments are assumed to be of type object. The domain of the object type is very large and too large, in practice, to search. Value constraints on scalar types are specified in terms of a set of intervals, to which the value must belong. In the case of the integer argument to Inc or Dec, for example, it must be a valid array index. ([1], [[0, 5]]) specifies a single interval. From within an interval, values are selected randomly with a uniform distribution. The integer that encodes the function must belong to [0, 3] but the tester decides that a uniform distribution is not suitable here, a no-op 0 should occur less frequently than a function call. In the specification ([1, 4, 2], [[0, 0], [1, 2], [3, 3]]), the tester specifies that a value is generated by selecting one of the three intervals with probabilities in the ratio [1: 4: 2] and then selecting from that interval. This mechanism can be used to specify non-uniform distributions that simulate the expected use profile or assist the search for test data. Value constraints are enforced not only when the random data is generated to seed the search but also when candidate tests are generated during the search by the search operators. This makes it easier to ensure that all inputs conform to the test program pre-conditions.

Test program analysis: The input file is parsed and if no syntax errors are found, an abstract syntax tree is created. A program dependency graph[9] is created for each function to be tested. JavaScript does not allow any unstructured transfers of control and so the program

dependency graph can be constructed by traversal of the abstract syntax tree[10]. The program dependency graph provides the control dependency conditions for each branch in the function under test. For branch coverage, these conditions form the basis of the test goal that the search process seeks to satisfy.

The data dependency graph is used to identify variable definitions that should be instrumented for data state instrumentation. Data state instrumentation is used as part of data diversity search strategy and for comparing the data states created in a mutant with that created in the original program.

Mutation tables: The current state of the prototype implements only behavioural mutations, i.e., substitution of operators and operands in statements. Object-oriented specific mutations[11] are not implemented. The abstract syntax tree nodes that represent operators and operands have additional attributes computed. Operators are associated with a table of replacement operators; operands are associated with a table of replacement operands. The replacement operand table holds all the in scope variables and literals from the program under test of compatible type. These mutation tables are constructed during traversal of the abstract syntax tree by collecting appropriate literals, variables and expressions. By systematically iterating through the mutation tables of each mutable object in turn, it is possible to generate all the mutations of the function under test.

Test program instrumentation: Depending on the coverage criterion and the progress of the test data search, various instrumented forms of the program under test are created. Predicate expressions in conditional statements are transformed as shown below to instrument for branch coverage:

```
if (CostEqInt(a[0] == a[n], tr)) {
return 1
```

CostEqInt is the cost function for comparing two integers, the cost value is stored in a trace object tr and a boolean is returned. Conditional statement trace objects accumulate branch cost data over multiple executions according to the scheme described in[12]. An assignment statement in which a variable of a basic type is defined may be associated with an assignment trace object. Assignment trace objects accumulate a frequency histogram of values assigned. This information is used to pursue a data-state diversity search strategy and also to compare data-states created when a test case is executed on the original function and on a mutant function.

Mutant generation: The abstract syntax tree together with the mutation tables is used as a meta-mutant[13] to generate mutant programs. The design of the meta-mutant involves a trade off between the speed with which successive mutants are generated and the speed with which any specific mutant executes. To arrive at a good compromise, it is important to consider that, broadly speaking; mutants can be classified as one of two kinds. One kind is easily killed, that is, killed by almost any test generated at random, such mutants we call soft mutants, the other kind is difficult to kill and they are usually executed many times during the search for a lethal test case, we call this kind hard. Fortunately, the vast majority of mutants are soft and so rarely need executing more than once. When the meta-mutant is generating soft mutants it is more important for the meta-mutant to generate them quickly than to generate mutants that execute quickly. Fortunately, very little instrumentation code need be inserted within a soft mutant. It is necessary to compare only the output of the subject program with that of the mutant. For speed of generation, soft mutants are not generated explicitly but emulated. Hard mutants, however, must execute quickly in spite of the additional instrumentation code that is inserted to guide the search for a lethal test case. Since mutant execution time is dominant, hard mutants are not emulated but generated explicitly as individual program that are compiled to machine code.

The hardness of a mutant, can of course, be determined only by trying to kill it. Since the vast majority of mutants are soft, initially all mutants are emulated. Those that survive the first test case that reaches the mutated statement are considered hard and thus generated as individual mutant programs with additional instrumentation to guide test data search.

In general, for mutation testing, the tool does the following:

- Produce mutants
- Randomly generate test input
- Run all mutants that have not been killed with some test input t, deciding which of these are killed to t
- State which mutants have yet to be killed

Test data search framework: The problem of taking a given program and constructing an input that produces a given program behaviour, is well known to be undecidable. Research effort has thus been directed towards heuristic approaches and a number of heuristic search methods have been investigated[3,5-8,20].

A genetic algorithm of the so-called steady-state variety such as Genitor[14] is the basic search technique

that is used to search for test data. For the purpose of the study presented here, a genetic algorithm may be described crudely in terms of three components, a set of candidate solutions, called a population, a cost function (also known as a fitness function) and a set of search (genetic) operators that can produce new candidate solutions by copying and modifying existing candidate solutions in the population.

The cost function evaluates each test case in terms of the search effort required to generate a solution from that test case. The selection of existing candidates is random but biased towards the most promising candidates as estimated by the cost function. The size of the population is usually fixed and so as new candidates are produced, the least promising are discarded. This is survival of the fittest. Over many iterations, the population is said to evolve towards a solution.

The guidance provided by the cost function is crucial to the success of the search. In the context of test data search, the candidate inputs are executed to establish if they contribute to the test goal. Typically, the candidate's inputs will not satisfy any test goal and so it is necessary to assess their utility in terms of generating test cases that do. The cost function uses information accumulated in the various trace objects of the instrumented program.

The test coverage criterion for the unit testing of a given program typically consists of a set of goals that must be satisfied. In the case of branch coverage, each branch to be executed is a goal. In the case of mutation analysis, the killing of a mutant is a goal. For each such goal, the search module eventually creates a genetic algorithm. Since the goal of executing a branch that is nested within a block controlled by an enclosing branch cannot be satisfied until the enclosing branch has been executed, a genetic algorithm for a branch or mutant is generated only when the predicate expression that controls that branch has been reached by some test case. At any stage in the search, a genetic algorithm search is pursued concurrently for all such goals.

All goals are pursued with equal resources and so each genetic algorithm is evolved for one cycle in turn. Whenever a genetic algorithm finds a test case to satisfy its goal, the genetic algorithm is deleted but the population of test cases is retained. Whenever a new genetic algorithm is created, the initial population of test cases it is seeded half randomly and half from all the existing test cases. These tests are evaluated by the cost function of the new genetic algorithm and the best tests accepted until the genetic algorithm population is full.

A multi-population genetic algorithm[15] extends the basic genetic algorithm by including a number of populations. Any genetic algorithm may create a number of populations as part of a search diversification strategy. The different populations within a genetic algorithm are intended to evolve different species that aim to satisfy the genetic algorithm goal but in different ways. In the context of branch coverage, the most general condition for execution of a branch is the control dependency condition for that branch. Initially, this condition is the basis of the cost function that evolves a single population. if after some time, a population is no longer evolving towards a solution, the control dependency condition is refined in different ways by adding additional branches (that do not conflict with the control dependency condition) that must be executed. These refined test goals are used to create cost functions for additional populations. In general, the population structure of a single genetic algorithm is a tree. Details are in[16].

In general, multi-population genetic algorithms may allow individuals to "migrate" from one population to another. Migration is normally limited in order to maintain the differences between populations. In the tool, the migration of test cases between populations is unrestricted. There are two reasons for this; firstly, each population has its own cost function which is the overriding determinant of which test cases remain in a population irrespective of the number of migrants from other populations. For this reason, unrestricted migration does not lead to the loss of diversity that it might in other multi-population genetic algorithms. Secondly, it is efficient to reuse executed tests wherever possible since the time required to execute the program under test is usually the most important factor that determines the speed with which test data is generated. Once a test case has been executed and the instrumentation data has been collected from the trace objects, an evaluation of the instrumentation data against any specific cost function can be produced relatively quickly.

RESULTS

Branch-coverage results: This tool is used to find test cases in order to satisfy different types of variables in branch coverage such as number, string and Boolean (flag problem).

This tool is used by the author in[12] which we focused on the test data generation to cover branches with string predicates.

We address in this study string equality, string ordering and regular expression matching. We applied a fitness function that depends on the string predicate.

Thus, for string equality we use the binary Hamming distance, character distance, edit distance and string ordinal distance, while for string ordering, the ordinal value method and single character pair ordering is applied.

The search for adequate test data is done using a GA. To improve the efficiency of the search, the input domain is restricted to characters within an ordinal range from 0 to 127. Further, the solution candidates are biased towards string literals that appear within the program under test. The experiment done in our study shows that the most effective result for string equality was obtained using the edit distance fitness function, while no significant difference was found in the fitness function for string ordering.

Mutation testing results: The most important functionality of the program would of course be to create mutants. Following explains how to do that.

The problem is reduced to mutate individual program elements, since a mutant normally differs from the program under test in one program element only.

Consider this statement in the program under test:

```
...
z = x + y;
...
```

How do we mutate this statement? One approach is to create a mutant which is one program containing all mutants. To declare which mutant is executing, an environment variable is set.

The mutant version of the above statement could be something like:

```
...
z = plusIntInt(x, y, 230, 232);
...
```

Each binary expression eligible for mutation is replaced with a function similar to the one above. The automatically generated plusIntInt function:

```
...
plusIntInt(int x, int y, int firstMut, int lastMut)
{
if (getCurrentMutation() >= firstmut &&
getCurrentMutation() <= lastmut)
{
if (getCurrentMutation() == firstmut)
return x - y;
if (getCurrentMutation() == firstmut + 1)
return x * y;
if (getCurrentMutation() == firstmut + 2)
return x / y;
return x + y;
}
else
return x + y;
}
...
```

checks whether, at this point in the program, it should execute a mutated statement. In the example, mutation number 230 mutates x+y into x-y, 231 into x*y and 232 into x/y. All other mutants executes the unmutated x+y.

Testing the triangle program: The benchmark program used to determine the type of a triangle; either it is illegal (the sides do not connect properly) or it is one of three valid cases scalene (no sides equal), isosceles (two sides equal) or equilateral (all sides equal).

Equivalent mutants: The problem with equivalent mutants still stands out as a time-consuming, error-prone and hence expensive task. To find all equivalent mutants in this specific case, we exhaustively tested every integer value in the domain:

$$D = (x, y, z)$$

where, x, y and z 2 [-20, 40]. Obviously, this method is infeasible in the general case.

Of the 117 mutants, these 9 (8 %) were found to be equivalent: 47, 81, 83, 96, 97, 99, 109, 110, 112. (Mutants 1 and 3 would be equivalent had the domain been limited to three integers; test case 13 tests with more and less parameters and kills those two mutants.) The numbers have no meaning other than identifying individual mutants.

Experiment: Mutation adequacy of a test set known to be adequate: Myers[23] lists 13 test cases that thoroughly test the triangle program in the appendix:

- A test case which represents a valid scalene triangle
- A test case which represents a valid equilateral triangle
- A test case which represents a valid isosceles triangle
- At least three test cases which represent valid isosceles Triangles such that you have tried all three permutations of two equal sides
- A test case in which one side is zero
- A test case in which one side is negative
- A test case with three positive integers such that the sum of two of them is equal to the third

- At least three test cases in category 7 such that you have tried all three permutations where the length of one side is equal to the sum of the lengths of the other two sides
- A test case with the sum of two of the numbers less than the third

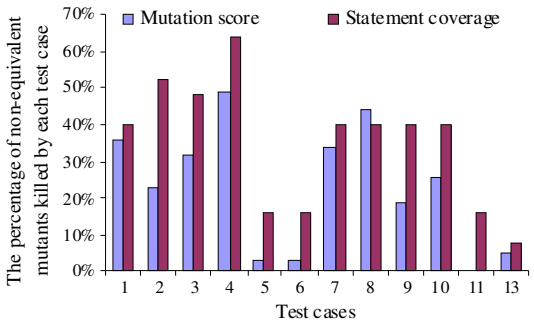

Fig. 2: The percentage of non-equivalent mutants killed by each test case. The total is the mutation score of the entire test set. The "statement coverage" column shows the percentage of executed statements of the program under test

- At least three cases in category 9 such that you have tried all three permutations
- A test case with all side lengths equal to zero
- At least one test case specifying non-integer values
- At least one test case specifying the wrong number of values (two or four)

Figure 2 shows the results of the test run. We can not draw any statistically valid conclusions based on this test run due to the limited number of mutants and the low complexity of our program under test. Yet, it is instructive to consider two things: that our test set indeed seem to be mutation adequate, although it could be better still and the correlation between statement coverage and mutation score.

To kill a mutant, it must be reached. If it is reached, the statement of that mutant is covered. Therefore, statement coverage must necessarily be a worse measure of test set adequacy.

DISCUSSION

The tool has been implemented by modifying the JScript compiler (written in C#) that is part of the SSCLI[17] distribution which implements the .NET framework. An abstract syntax tree walker was written in order to implement a number of abstract syntax tree operations which include, program dependency graph construction, various transformations for program instrumentation, collection of various lexical elements, literals, variables, to construct mutation tables. The .NET framework provides reflection from which it is possible to extract information about the program under test. Given access to the abstract syntax tree, no use is made of reflection to analyse the program under test.

The test tool does not generate test data entirely automatically. Many script programs execute in a complex context which contains large complex objects. For example, the context of a script program may include the browser in which it is executing, the window, various user interface forms, a word document and so on. The tester is still responsible for creating a significant part of this context. The test tool provides the tester with a tool to search for parameter values and statement selection and sequencing.

The tool is a prototype and currently, a number of the JavaScript language constructs cannot be handled. Programs that contain these constructs cannot be tested.

In general, this testing tool is generally laboratory prototypes. I am not aware of any fully featured commercial tools for testing.

CONCLUSION

The test data generation tool is able to generate test data for JavaScript functions. The tool may be directed to generate branch coverage data or mutation coverage data. The tester is responsible for ensuring that the function under test executes in an appropriate context ,but the tool will search for parameter values and statement selection and sequencing. Requiring the tester to providing an appropriate context means that although test generation is not entirely automatic, it is a practical tool.

This is significant because complete automation will reduce the cost of software testing dramatically and also facilitate continuous testing. It is reported that at least 50% of the total software development costs is due to testing, and 10–15% of development time is wasted due to frequent stops for regression testing. Automation will also help get rid of cognitive biases that have been found in human testers.

A graphical user interface is to be added to allow the tester to add test cases manually, execute the test cases and to obtain path and data state information. The tester will be able to modify and re-execute the test.

ACKNOWLEDGMENT

The researcher would like to express their gratitude to the anonymous referees for their valuable and helpful comments and suggestions in improving the study.

REFERENCES

1. DeMillo, R., 1978. A probabilistic remark on algebraic program testing. Process. Lett., 7: 193-195.
2. Hamlet, R., 1977. Testing programs with the aid of a compiler. IEEE Trans. Software Eng., 3: 279-290. DOI: 10.1109/TSE.1977.231145
3. Jones, B., R.H. Sthame and D. Eyres, 1996. Automatic structural testing using genetic algorithms. Software Eng. J., 11: 299-306.
4. Korel, B., 1990. Dynamic method for software test data generation. Software Test. Verificat. Reliabil., 2: 193-213.
5. Tracey, N., J. Clark and K. Mander, 1998. Automated program flaw finding using simulated annealing. Software Eng. Notes, 23: 73-81.
6. Tracey, N., J. Clark, K. Mander and J. McDermid, 1900. Automated test data generation for exception conditions. Software Pract. Exp., 30: 61-79. DOI:10.1002/(SICI)1097024X(200001)30:1<61::AID-SPE292>3.0.CO;2-9 .
7. Baresel, A., J. Wegener and H. Sthamer, 2001. Evolutionary test environment for automatic structural testing. Inform. Software Technol., 43: 841-854.
8. Baresel, A., H. Sthamer and M. Schmidt, 2002. Fitness Function Design to Improve Evolutionary Structural Testing. Proceedings of the Conference on Genetic and Evolutionary Computation, Morgan Kaufmann, New York, USA, July 9-13, pp: 1329-1336.
9. Ferrante, J., K. Ottenstein and J. Warren, 1987. The program dependency graph and its use in optimization. ACM Trans. Programm. Languages Syst., 9: 319-349. DOI: http://doi.acm.org/10.1145/24039.24041
10. Byers, D., M. Kamkar and T. Palsson, 2001. Syntax-directed construction of value dependence graphs. Proceeding of the IEEE International Conference on Software Maintenance, Florence, Italy, Nov. 7-9, pp: 692-703. DOI: 10.1109/ICSM.2001.972788.
11. Buy, U., A. Orso and M. Pezz´e, 2000. Automated testing of classes. Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Aug. 22-25, ACM Press, Portland, OR., USA., pp: 39-48. DOI: http://doi.acm.org/10.1145/347636.348870.
12. Alshraideh, M and L. Botacci, 2006. Automatic software test data generation for string data using heuristic search with domain specific search operators. Software Test. Verificat. Reliabil., 16: 175-203. DOI: 10.1002/stvr.354.
13. Untch, R., 1993. Mutation analysis using mutant schemata. Proceeding of the International Symposium on Software Testing and Analysis, June 28-30, ACM Press, Cambridge MA., New York, USA., pp: 139-147. DOI: http://doi.acm.org/10.1145/174146.154265.
14. Whitley, D., 1989. The Genitor Algorithm and Selective Pressure: Why rank-based allocation of reproductive trials is best. Proceedings of the 3rd International Conference on Genetic Algorithms, J. David Schaffer, George Mason University, Fairfax, Virginia, USA, June 4-7, pp: 116-121. http://www.cs.colostate.edu/~genitor/1989/ranking89.ps.gz
15. Cantu-Paz, E., 1988. A survey of parallel genetic algorithms. Calculat. Parall. Reseaux ET Syst. Rep., 10: 141-171.
16. Bottaci, L., 2005. Use of branch cost functions to diversify the search for test data. Proceedings of the Workshop on UK Software Testing, Sep. 5-6, University of Sheffield, UK. pp: 151-163. www.dcs.hull.ac.uk/people/csslb/doc/published/uk**test**05/**branch**diversity.pdf.
17. Stutz, D., T. Neward and G. Shilling, 2003. Shared Source CLI Essentials. 1st Edn., O'Reilly and Associates, Inc., Sebastopol, CA., USA. ISBN: 059600351X.
18. Meudec, C., 2001. ATGen: Automatic test data generation using constraint logic programming and symbolic execution. J. Software Test. Verificat. Reliabil., 11: 81-96. DOI: 10.1002/stvr.225.
19. Gupta, N., A. Mathur and M.L. Sofia, 2000. Generating test data for branch coverage. Proceedings of 15th IEEE International Conference on Automated Software Engineering, September, Grenoble, France, pp: 219-228. DOI: 10.1109/ASE.2000.873666.
20. Marciniak, J.J., 2002. Encyclopaedia of Software Engineering. 2nd Edn., Wiley, New York, USA., pp: 1327-1358. ISBN: 978-0-471-37737-5.
21. Whittaker, J.A., 2000. What is software testing? And why is it so hard? IEEE Software, vol. 17, no.1,pp.70-79. DOI:10.1109/52.819971
22. Chang, K., J. Cross, W. Carlisle and S. Liao, 1996. A performance evaluation of heuristics_based test case generation methods for software branch coverage. Int. J. Software Eng. Knowl. Eng., 6: 585-608.
23. Myers, G.J., 2004. The Art of Software Testing. 2nd Edition, Wiley-Interscience, New York, ISBN: 978-0-471-46912-4, pp. 156.