DYNAMIC SEARCH-BASED TEST DATA GENERATION FOCUSED ON DATA FLOW PATHS

Anastasis A. Sofokleous and Andreas S. Andreou
Department of Computer Science, University of Cyprus, 75 Kallipoleos Str., Nicosia, Cyprus

Keywords: Software Testing, Control Flow Graph, Data Flow Graph.

Abstract: Test data generation approaches produce sequences of input values until they determine a set of test cases that can test adequately the program under testing. This paper focuses on a search-based test data generation algorithm. It proposes a dynamic software testing framework which employs a specially designed genetic algorithm and utilises both control flow and data flow graphs, the former as a code coverage tool, whereas the latter for extracting data flow paths, to determine near to optimum set of test cases according to data flow criteria. Experimental results carried out on a pool of standard benchmark programs demonstrate the high performance and efficiency of the proposed approach, which are significantly better compared to related search-based test data generation methods.

1 INTRODUCTION

As research focuses on software testing, studies show that this process is one of the key-attributes for delivery high quality end-systems within time and cost constraints. Existing challenges in this area involve the development of automatic software testing methods that can test, or generate the test data in order to test a program (McMinn, 2004). Testing adequacy, i.e. the effectiveness of a testing process on a program, as well as the testing termination criterion, i.e. when the testing process should be terminated, is determined by certain coverage criteria. Among these the most common are the control and data flow criteria, with the former being currently the most widely used (Kapfhammer, 2004).

This paper extends previous work described in Sofokleous and Andreou (2007) where a dynamic testing framework based on control flow graphs has been proposed and demonstrated. In this work the framework has been enhanced and now consists of a program analyser and a test case generator; the former analyses programs, creates control and data flow graphs, and evaluates test cases in terms of testing coverage. The test cases generator utilises a specially designed GA to generate test cases with respect to a coverage criterion. The GA's fitness function is guided by characteristics of the data flow graph of the program under test.

The contribution of this work may be summarised to the following.

- It proposes a new test data generation scheme based on evolutionary computing, which is simple, practical and fast. It integrates two systems for analysing the program under testing and determining the required set of test cases.
- One of the novelties of this work is the integration of the control flow graph and data flow graphs. The program analyser utilises a dynamic code coverage module that determines the executed code directly on the control flow graph; in addition, it utilises the corresponding data flow graph with the related criteria for generating and evaluating test data. In this paper we use the *All_DU_Paths* data flow criterion.
- The testing approach uses a novel mutation operator for mutating composite genes.
- All these are encapsulated in a prototype proof of concept application, with which we performed experiments on a pool of standard programs. Preliminary results show that this approach outperforms other related studies which generate test data in relation to data flow criteria.

A. Sofokleous A. and S. Andreou A. (2008).
DYNAMIC SEARCH-BASED TEST DATA GENERATION FOCUSED ON DATA FLOW PATHS.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - AIDSS*, pages 27-35
DOI: 10.5220/0001692600270035

The rest of the paper is organised as follows. Section 2 presents related work including open challenges on the subject, section 3 describes the proposed testing framework and section 4 presents the experimental results. Section 5 concludes and suggests some steps for future work.

2 RELATED RESEARCH

Our approach focuses on dynamic software test cases generation, a search-based technique that uses feedback to adapt its behaviour and determine an adequate set of test cases according to a testing coverage criterion. Research literature classifies search based techniques to random, if it generates test data randomly, or dynamic, if it considers the results produced to adapt the testing processing (Korel, 1996; Michael et al., 2001). Studies have showed that random based-approaches, which are less resource consuming compared to dynamic ones, cannot determine efficiently the required test cases for complex programs (Korel, 1996).

The most popular dynamic-based approaches utilise genetic algorithms for generating test cases, e.g. see Michael et al. (2001) and Michael and McGraw (1998). To evaluate the efficiency of the generated results, and guide the algorithm during searching, researchers and practitioners use testing coverage criteria, the most popular of which are Control Flow and Data Flow based criteria (Kapfhammer, 2004; Zhu et al., 1997). Thus, dynamic testing systems repeat a testing cycle of three main steps: (i) generate *n* test cases, (ii) utilise a coverage tool to evaluate each test case with respect to a coverage criterion and (iii) use the results to guide the next iteration. The use of a control flow criterion, e.g. statement or edge, implies analysis of the program's structure, e.g. branches and loops, whereas for a data flow criterion, e.g. all-edges and all-uses, it is necessary to examine the data part of the program, e.g. variable to value bounding relation and variable usage within the program. Examples of control flow criteria may be found in Andrews et al. (2006) and Zhao (2003). This paper focuses on test data generation using data flow criteria. The most relevant research studies explore the definition and usage of data flow coverage criteria which yield testing results comparable to those of control flow criteria, in terms of testing adequacy.

Laski and Corel (1983) propose a strategy that uses the definition – *use* chain of a variable in order to guide the program testing. This approach defines two different criteria, which are both based on the observation that on any given node there might be uses of *z* variables, *z>1*, on which definition is made on previous *z* nodes. A definition at *node-i* is considered to be live at a *node-j*, if there are no redefinitions of this variable between *node-i* and *node-j*. The first criterion requires that each use of the variable in nodes where the definition is "live" is tested at least once. The second criterion requires that each elementary data context of every instruction is tested at least once. The elementary data context of an instruction *k* is defined as the set of definitions *D(k)* for the variables of *k*, such that there exists a path from the beginning of the program to *k*, where the definitions *D(k)* are live when the path reaches *k*. The authors also propose the modified version where each ordered elementary data context is tested at least once. A detailed presentation of the criteria can be found in Laski and Korel (1983), while improved definitions are given in Clarke et al. (1989).

Ntafos also proposed a method for selecting paths, namely *k-dr* interactions (Ntafos, 1984; Ntafos, 1981). Interactions between different variables are captured in terms of alternating definitions and uses, called *k-dr* interactions. Variable x_1 is defined at node n_1 and then used in node n_2. At node n_2, variable x_2 is defined and then used in node n_3, where a third variable is defined. This sequence of definitions and uses can be noted as $\left[d_{n_1}^{x_1}, u_{n_1}^{x_1}, d_{n_2}^{x_2}, u_{n_2}^{x_2}, d_{n_3}^{x_3}, u_{n_{13}}^{x_3}, ..., d_{n_m}^{x_m}, u_{n_m}^{x_m}\right]$.

Note that between each definition and use for a variable, the path is *def-clear*. Such a sequence of *k-1 du-pairs*, *k>1*, is called *k-dr* (definition/reach) (Ntafos, 1984). For the latter, it is necessary to test *dr* interactions of specific length.

Rapps and Weyuker's idea for path selection criteria is clearly derived from the set of criteria defined for control flow graphs and were originally defined in Rapps and Weyuker (1982). Starting by redefining *all-paths*, *all-edges* and *all-nodes* criteria, they extend the set of criteria by defining *all-defs*, *all-uses*, *all-c-uses*/*some-p-uses*, *all-p-uses*/*some-c-uses*, *all-p-uses* and finally *All-DU-Paths*. The *all-c-uses* criterion was added later on in the list of the aforementioned criteria (Frankl and Weyuker, 1988). The whole set is based on the definitions and uses of

the variable but uses are distinguished in *c-uses* and *p-uses*. The first term is used to define a use in a computation (at the right hand side of an assignment) and the second to define the use of the variable as a predicate in a Boolean calculation. The criteria are analytically presented in Clarke et al., (1989) and Rapps and Weyuker (1982). Rapps and Weyuker (1985) provide the "hierarchy" of the criteria with a robust proof. Data flow criteria were examined by different research teams from time to time, aiming at defining a partial order between all criteria or revealing their weaknesses and strengths, e.g. see Clarke et al. (1989) and Ntafos (1988)).

This paper proposes a framework that uses genetic algorithms for searching the input space and determining a set of test cases for a program under testing. The genetic algorithm encodes the test inputs as genes and evolves test cases targeting specific paths. The paths are extracted according to a data flow graph criterion, the *All-DU-Paths*. Thus, by generating test data for each extracted path, the framework can achieve testing coverage according to the *All-DU-Paths* data flow graph criterion. This criterion will form the basis of the computational intelligent part of the testing framework and more specifically it will guide our genetic algorithm to evolve appropriate test data so as to achieve the highest possible coverage. This guidance is embedded in the fitness function of the algorithm as will be explained later on. In addition, in this paper we propose the use of intra-mutation genetic operator to mutate internally a gene, if it encodes a collection of items (e.g. an array) or an object.

3 THE TESTING FRAMEWORK

Our testing framework consists of a program analysis system, which analyses the code of a program under test and creates the control flow and data flow graphs, and a test case generation system that generates test cases using the program analysis system and based on each generation's feedback. The design and usage of the two systems are explained in greater detail in the next sections.

3.1 The Program Analysis System

The program analysis system consists mainly of the *static* and *dynamic* analysis sub-systems; the former performs non-runtime analysis, i.e. without executing the program under study, while the latter simulates runtime behaviour, i.e. it simulates the execution of a program based on a pair of input values.

The static module parses Java code, e.g. a class, and creates code representations, such as control flow and data flow graphs. Currently, this module first parses the program under test, then uses a module to visit each block of the program and build the control flow graphs, a graph for each method. A control flow graph uses nodes and edges to represent the statement and the flow of the program code, respectively. However, a control flow graph captures only the flow of a method and each call to another method is shown as a call to a black-box, i.e. it takes input and provides output, which can be expanded upon request to another control flow graph. Then, each control flow graph is used to build its respective data flow graph, which presents the data flow and statements using nodes and edges, respectively.

The dynamic analysis sub-system is mainly responsible for the runtime evaluation of the program code. Currently, this sub-system employs a dynamic code coverage module, which, compared to other code coverage tools, is able to determine the code coverage directly on the control flow graph, i.e. without using the program code. Specifically, using a pair of input values it can simulate the execution of the program under testing; thus, by evaluating the expressions of each vertex and following the directed edges, it can determine the executed code. The executed code is illustrated graphically on the control flow graph. In addition to the executed code, by selecting a path, the code coverage can determine how close a test case is, to executing this path. This allows the testing process to perform focused searching as we will see later on.

The program analysis system advertises its functionality through an API, which can be used by other systems, such as the test case generation system. Such systems can utilise the program analysis system to obtain program information (e.g. variables types and usage, scope of variables), determine part of the code on a code representation (e.g. the control flow or data flow graphs), determine the coverage for a pair of input values, etc. In this paper, the test case generation system utilises the control flow-based code coverage module alongside with the data flow graph in order to assess the testing coverage of the program under test with respect to the *All-DU-Paths*.

3.2 The Test Cases Generation System

The test data generation system houses a specially designed genetic algorithm, which, with the aid of the program analysis system, can determine a near to optimum set of test cases based on a certain coverage criterion.

Initially, the program analysis system analyses the program under testing: it parses the program, determines the units of testing, for each of which it creates a control flow graph and a data flow graph (Sofokleous and Andreou, 2007; Sofokleous et al., 2006). Focusing on each testing unit, the test data generation system uses the data flow graph to extract the *All-DU-Paths*, each representing a target to be tested. Then, the algorithm visits the I^{th} target path denoted as P_I, initially *I=1*, and initiates the genetic algorithm (GA); the GA aims to produce a test case that can exercise each node in path P_I.

GAs have been widely used as optimization techniques. By maintaining and evolving a population of candidate solutions, GAs may determine a near to optimal solution. The evolution takes place through generations by mimicking natural evolution, that is, through crossover and mutation of each generation's population. The fitness of each individual solution is calculated using special designed functions which capture the optimization constraint of each problem. GAs are used when the search input is huge and therefore evaluating each solution one by one is not feasible. Thus, instead of evaluating every solution in the search space, GAs' design allows the direction to the optimal solution by only evaluating samples of the solution space. A value reflects how close an individual solution is to the optimum one; another advantage of GAs is that they allow evaluating the solution with respect to the rest of the population, i.e. the value assigned to each solution depends on its content and on what are the rest of the solutions in the set. This paper uses GAs to address the problem of generating test cases, a problem that encounters similar challenges to the aforementioned. For example, in our case the search space, which is the domain of the input space, is huge and therefore GA can assist in evaluating only a sample of the domain space and directing the search to a near to optimal solution.

In this paper, a GA chromosome describes a test case and a gene encodes an input parameter of the testing unit. For example, if a method has three input parameters, say *x*, *y* and *z*, then each chromosome will have three genes, where each gene will encode one of these parameters. The design of the chromosome is illustrated in figure 1. A gene, which is implemented as a data structure, includes the type of the input variable and the initial value (input value). In addition, each chromosome's fitness value reflects the value of the test case it offers, which is described in the following paragraphs.

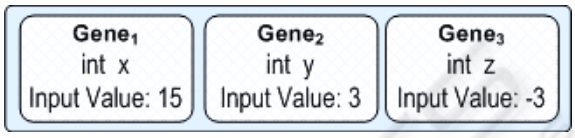

Figure 1: The Chromosome Design.

The fitness function is expressed as follows:

$$f(C_K, P_I) = \#nodes_{exec} + dist\left(node_j\right) \quad (1)$$

where C_k is a chromosome that contains the TC_K test case, $\#nodes_{exec}$ is the number of exercised nodes with respect to TC_K, and $dist\left(node_j\right) = 0$ if TC_K exercises every node of P_I, otherwise $0 < dist\left(node_j\right) < 1$ if using TC_K cannot pass $node_j$ and hence this value describes how close TC_K is to pass $node_j$. Part of our encoding scheme resembles the one reported in Michael et al. (2001); the authors of this study evaluate their chromosomes using the distance approach but only using the control flow graph. Our main differentiation, however, is that the searching and evaluation takes into account not only the control flow graph but also each path of the data flow graph. This way the search of a test case for a target path is more focused as it targets the nodes of a path and hence it is not biased by the chromosomes that can exercise nodes of other paths.

To evaluate chromosome C_k, GA passes its content, i.e. test case TC_K, to the coverage module. The coverage module, which runs dynamically on the control flow graph, determines the executed control flow nodes, say $\mathrm{N}_{\mathrm{CFG}}$. Suppose the data flow path P_I is the target path, then $\#nodes_{exec}$ is the number of P_I nodes which have been executed on the control flow graph, i.e.

$\#nodes_{exec} = \left|\mathrm{N}_{\mathrm{CFG}} \bigcap P_I\right|$. If P_I consists of *n* nodes, *n>1*, and $\#nodes_{exec} < n$, it means that TC_K covered only partially the path and that there is one or more predicate conditions at another node, e.g. at $node_j$, that prohibits the execution flow from traversing from $node_j$ to some of its successor nodes. Thus, to direct the search towards the right test case that can satisfy $node_j$'s condition(s), we add an extra value, i.e. $dist\left(node_j\right)$, the of which role is twofold: (i) to show how good TC_K is compared to the rest of the chromosomes that failed at node $node_j$, and (ii) to show how close TC_K is for successfully traversing $node_j$. For example, $x > y$ is the condition of $node_j$ that has to be evaluated to false then

$$dist\left(node_j\right) = \begin{cases} x - y, & if\ x > y \\ 0, & if\ x \le y \end{cases} \quad (2)$$

Thus, the distance is progressively reduced as x approaches y, and becomes equal to zero if its value becomes equal or lower than y as it evaluates the condition $(x > y)$ to the desired value.

At each generation, the population is evolved by repeating a cycle of evaluating the population, selecting a pool of solutions (with respect to the evaluation results), and applying genetic operations, such as the mutation and crossover, on the selected pool of solutions. The crossover operation selects two chromosomes and swaps internal parts cut at a selected crossover point and produces offspring chromosomes. Offspring chromosomes are added to the list of chromosomes that pass to the next generation. Likewise, mutation operation mutates a gene: if a chromosome is selected for mutation, then the algorithm selects one or more genes to be replaced with new genes. A common mutation operation generates a new gene as a clone of the selected gene with new random values. In this paper, for primitive types, such as integer and float, the mutation operation generates the new values using a stepwise approach, e.g. the new value of a variable is $x \pm random(\min V, \max V)$, where *x* is the current value and $random(\min V, \max V)$ returns a number in the range defined by a minimum and a maximum value specific for that variable . The algorithm uses a crossover and mutation probability that define the likelihood that a chromosome is selected for crossover and mutation, respectively, and a mutation step probability which swaps between small mutation steps (e.g. $\min V = 2$, $\max V = 2$) and large steps (e.g. $\min V = \infty$, $\max V = \infty$).

As discussed above, a gene is an input variable while a chromosome is a set of genes and therefore it encodes a complete test case. A parameter can be of a primitive type, such as integer or float, an array or an object. Most of the test case generation approaches use input variables of either primitive types (integers or/and floats) or of arrays of primitive types (array of integers/floats). Thus, usually a gene encodes a primitive type, and in some cases, an array. However, program flow in the latter case depends not only on the values of the array but also on its size. A conventional mutation operator replaces the entire array with a new one; in this paper we propose a new type of mutation operator, which can act internally on composite genes. Thus, instead of only mutating the entire gene, the new operator may select and mutate one or more internal elements of the gene (see figure 2). For example, if a gene encodes an array of integers, then the mutation can either mutate the whole gene, which implies a new array (size and values of the array may vary), or mutate only a specific value of the array, which leaves the size of the array and the rest of the values

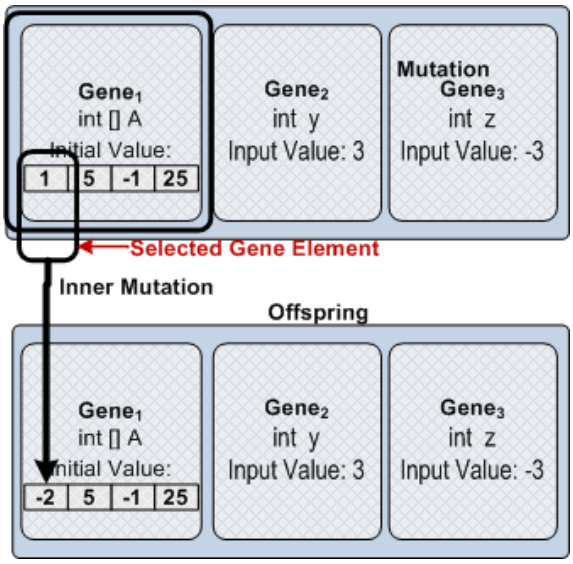

Figure 2: Inner Mutation for composite Genes.

unchanged. For this operation we use the mutation-switch probability rate which determines if the mutation will mutate the entire gene or part of the gene, while for the latter we use an additional probability to determine the part of the gene to be mutated.

Figure 3 shows the proof of concept prototype application, which consists of the program analysis system, the test case generation system and a graphical user interface with which a user can set the configuration parameters and interact with the control flow and data flow graphs. Initially, the user activates the program analysis system for a program under testing; then the user can switch between the control flow graph and the data flow graph. Selecting to generate test cases, a pop-up dialog requests the parameters pertinent to the testing process; this includes the selection of a testing coverage criterion (e.g. control flow or data flow criteria) and the definition of the population size, the probabilities of the operators (mutation rate, crossover rate, the mutation step rate, the rate for switching between inner and outer mutation, etc). Also the selection operator (tournament or Roulette Wheel) is defined here, along with the maximum number of generations. Then, the algorithm generates test cases until it either achieves full coverage according to the *All-DU-Paths* criterion or reaches the maximum number of generations. The selected test cases are presented in a grid, whereas selecting a test case from the grid, triggers the graphical depiction of executed nodes (it does so by using a different colour) on the control flow graph.

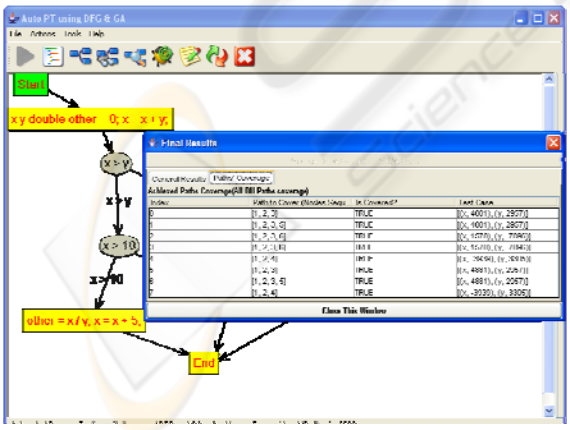

Figure 3: Prototype Application.

4 EXPERIMENTAL RESULTS

We evaluated the performance of the proposed framework on a pool of standard programs, which have been also used as benchmarks by other testing methods, most of which utilise on control flow graph criteria, e.g. see Michael et al. (2001). This pool of programs includes the Binary Search, the Bubble Sort, the Insertion Sort, the Quadratic formula solving, the Triangle Classification and the factorial program. Based on the best results of a preliminary empirical investigation, we set the GA's population size equal to 100 chromosomes, the probabilities of crossover between 0.40 and 0.50, of mutation between 0.05 and 0.15, of switch-mutation step switching to 0.50, and the maximum generation number to 2000. The Roulette Wheel was defined as the selection operator and also the feature of elitism was activated, that is, the algorithm always passes the best chromosome unchangeable to the next generation.

Basically the testing system invokes a new GA for each path extracted according to the *All-DU-Paths* data flow criterion. Thus, the algorithm terminates when all GAs terminate, where a GA terminates either when it determines a solution for its objective (i.e. a test case that can cover the target path) or when it reaches the maximum number for generations. Table 1 compares the coverage ability of four testing generation algorithms applied on 6 standard programs. The algorithms are: the random, the gradient decent, a standard genetic algorithm which calculates the fitness function as a function of the executed nodes on the complete control flow graph and our approach as described in this paper. The results show that our approach can achieve full coverage for all of the programs tested, while the rest of the algorithms present weakness mostly in the quadratic formula and the triangle classification, with the best results confined to achieving less than 90% coverage.

Table 1: Testing Coverage Comparison of test data generation algorithms on a pool of standard programs.

Algorithm / Program	Random	Gradient Decent	GA-1	Our Approach
Binary Search	78	100	77	100
Bubble Sort	100	100	100	100
Insertion Sort	100	100	100	100
Quadratic Formula	74	70	73	100
Triangle Classification	85	75	85	100
Factorial	100	100	100	100

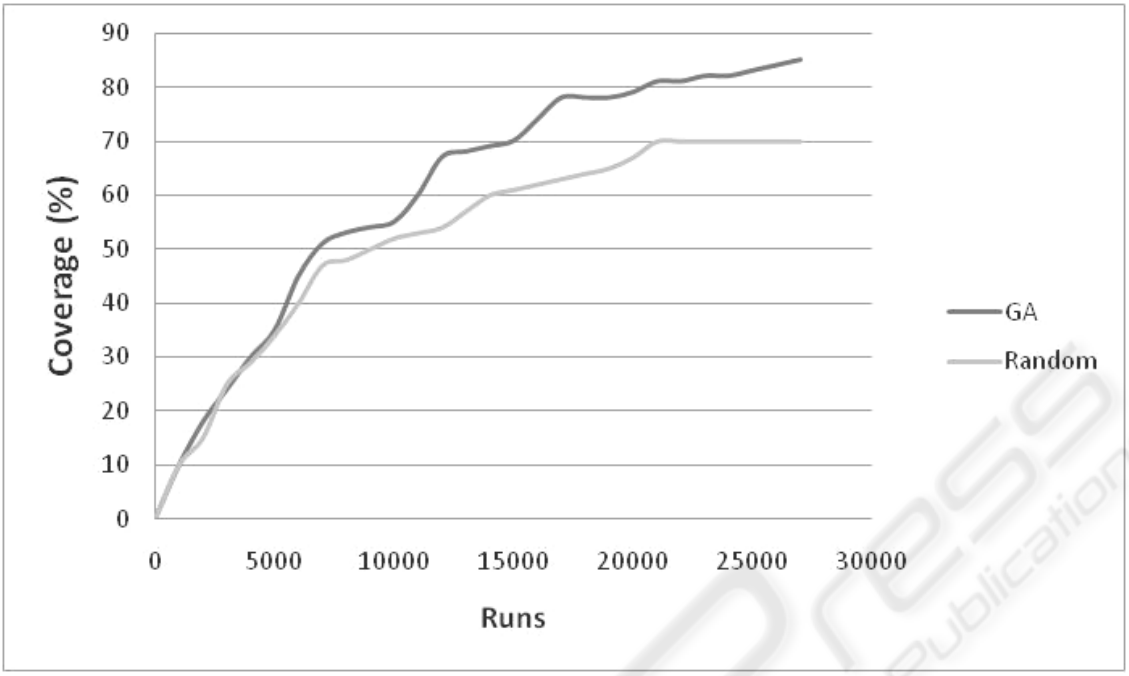

Figure 4: Performance over time.

Table 2 presents experiments conducted on a set of Java programs varying in size and complexity. The complexity is determined based on conditions, e.g. complexity is high (H) if there are conditions consisting of three or more predicates, medium (M) if predicates are only two, and simple (S) denotes conditions with only one predicate. Table 2 describes LOC (lines of code), #TC (the size of the return set of test cases), the complexity of the program, the testing coverage in relation to the *All-DU-Paths* data flow criterion and the number of runs, where a run is the simulation of a test case so as to determine its coverage. It is evident that the proposed approach is highly successful even in cases with large programs of considerable complexity.

Table 2: Empirical results on a pool of sample programs varying in both size and complexity.

ID	LOC	# TC	Complexity	Coverage	#Runs
1	20	2	S	100%	2000
2	20	4	M	100%	2000
3	20	4	H	100%	2000
4	50	4	S	100%	2000
5	50	8	M	100%	2000
6	100	7	S	100%	2000
7	100	7	S	100%	2000
8	250	9	M	100%	3500
9	250	10	M	100%	6400
10	500	12	S	100%	12400
11	1000	15	M	94%	16300
12	1500	23	M	90%	19500

Figure 4 presents a graphical comparison of our approach against the random algorithm on a randomly generated program of 2000 LOC. The figure shows performance over time, where performance is measured by the corresponding testing coverage percentage and time is shown as a number of runs.

The preliminary results reported show that the testing framework is efficient and capable of testing programs with respect to data flow criteria.

Compared to other testing approaches, such as the random and the conventional usage of genetic algorithm (e.g. using the complete control flow graph for evaluating test cases and directing the search), our proposition can achieve better coverage, in shorter execution time and can test more complex programs.

5 CONCLUSIONS

This paper described a search based test case generation approach which uses both the control flow and data flow graphs of a program under test

for searching and determining a near to optimal set of test cases according to the data flow criterion. Based on previous work, we extended a test case generation system by integrating and exploiting the functionality of the two graphs. Control flow graph assists in dynamic code coverage, i.e. determines executed code on the control flow on the fly. Data flow graphs assist in extracting the data flow paths according to the *All_DU_Paths* data flow criterion. A test case generator uses the control flow graph to identify the executed part of an *All_DU_Path* and evaluate the test case. The test case generator employs a genetic algorithm which uses the feedback to adapt its behaviour and hence to come closer to the appropriate set of test cases. In this work, we also addressed the challenge of the composite gene mutation and we proposed the switch-mutation operation, which can mutate not only the gene (i.e. a variable of the test case) as a complete unit but also an element of the gene (i.e. an element of an array which is treated as an input variable). Experimental results revealed the efficiency of our approach over a pool of standard programs. These results indicated better performance compared to similar studies, both in terms of testing coverage and execution time, the latter being calculated in terms of evaluation runs.

Currently we are carrying out more experiments with larger and more complex programs which can assess the performance of our framework in more realistic environments. Future work will consider the extension of the framework to support control flow graph slicing so as to identify faulty parts of the code. Future work will consider the modification of the new mutation operator to support composite genes of objects, i.e. when one or more objects are given as input variables to a testing unit. In this case, the test case generator may need to exercise the following: (i) the object itself, e.g. by generating a new object, (ii) the state of the object, e.g. by calling some methods of the object before assigning it to the gene and (iii) a descendent object (following heritage tree) and hence the late binding, by creating an object of a sub-class and up-casting it to the target object. Finally, we plan to investigate the incorporation of a new type of graph which will be able to capture both the control flow graph and the object oriented features.

REFERENCES

Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32 (8), 608-624.

Clarke, L.A., Podgurski, A., Richardson, D.J., Zeil, S.J. 1989. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering* 15 (11), 1318-1332.

Frankl, P.G., Weyuker, E.J. 1988. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering* 14 (10), 1483-1498.

Kapfhammer, G.M. 2004. Software testing. In: Tucker, A.B. (Ed.), CRC Press, Boca Raton, FL, 105.1-105.44.

Korel, B. 1996. Automated test data generation for programs with procedures. In: *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis* , San Diego, California, United States, 209-215.

Laski, J.W., Korel, B. 1983. Data flow oriented program testing strategy. *IEEE Transactions on Software Engineering* 9 (3), 347-354.

McMinn, P. 2004. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability* 14 (2), 105-156.

Michael, C.C., McGraw, G., Schatz, M.A. 2001. Generating software test data by evolution. *IEEE Transactions on Software Engineering* 27 (12), 1085-1110.

Michael, C., McGraw, G. 1998. Automated software testdata generation for complex programs. In: *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, Honolulu, Hawaii, October 1998, 136-146.

Ntafos, S.C. 1984. On required element testing. *IEEE Transactions on Software Engineering* 10 (6), 795-803.

Ntafos, S.C. 1988. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering* 14 (6), 868-874.

Ntafos, S.C. 1981. On testing with required elements. In: *Proceedings of IEEE-CS COMPSAC*, November 1981, 132-139.

Rapps, S., Weyuker, E.J. 1982. Data flow analysis techniques for test data selection. In: *Proceedings of the 6th IEEE-CS International Conference on Software engineering*, Tokyo, Japan, September 1982, 272-278.

Rapps, S., Weyuker, E.J. 1985. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering* 11 (4), 367-375.

Sofokleous, A., Andreou, A. 2007. Batch-optimistic test-cases generation using genetic algorithms. In: Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), Patra, Greece, October, 157-164.

Sofokleous, A.A., Andreou, A.S., Ioakim, G. 2006. Creating and manipulating control flow graphs with multilevel grouping and code coverage. In: Proceedings of the 8th International Conference on Enterprise Information Systems (ICEIS 2006), Paphos, Cyprus, May 2006, 259-262.

Zhao, J. 2003. Data-flow-based unit testing of aspect-oriented programs. In: *Proceedings of the 27th IEEE-CS Annual International Conference on Computer Software and Applications (COMPSAC '03)*, Dallas, Texas, USA, 188-197.

Zhu, H., Hall, P., May, J. 1997. Software unit test coverage and adequacy. *ACM Computing Surveys* 29 (4), 366-427.