An automatic software test-data generation scheme based on data flow criteria and genetic algorithms

Andreas S. Andreou
Department of Computer Science, University of Cyprus
aandreou@cs.ucy.ac.cy

Kypros A. Economides
Department of Computer Science, University of Cyprus
cspgok2@cs.ucy.ac.cy

Anastasis A. Sofokleous
School of information computing and Mathematics, Brunel University, UK
anastasis.sofokleous@brunel.ac.uk

Abstract

Software test-data generation research primarily focuses on using control flow graphs for producing an optimum set of test cases. This paper proposes the integration of a data flow graph module with an existing testing framework and the utilisation of a specially designed genetic algorithm for automatically generating test cases based on data flow coverage criteria. The enhanced framework aims to explore promising aspects of software testing that have not yet received adequate research attention, by exploiting the data information of a program and provide a different testing coverage approach compared to existing control flow-oriented ones. The performance of the proposed approach is assessed and validated on a number of sample programs of different levels of size and complexity. The associated experimental results indicate successful performance in terms of testing coverage, which is significantly better when compared to those of existing dynamic data flow-oriented test data generation methods.

1. Introduction

Recent research studies focus on automatic generation of optimum sets of test cases for low-cost, high quality, unbiased and effortless software testing. Automatic software testing is either static or dynamic; the former approach uses the static information of the internal structure of programs, while the latter utilizes the runtime information of programs through execution with parametric input values [1]. To measure testing adequacy, i.e. how well a program is tested by a set of test cases, many researchers propose the use of different coverage criteria (e.g. control flow criteria) [2]. In addition, testing coverage criteria usually define the termination condition of the testing process.

This paper builds upon and extends previous work on test data generation [3, 4] where a dynamic testing framework based on control flow graphs was proposed and demonstrated. This framework is enhanced in this work with a computational intelligent part that couples data flow graphs with genetic algorithms (GA) to generate optimum test cases. The framework 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 special purpose 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 innovation of this work may be summed up to the following:

- Proposition of a new test data generation scheme based on evolutionary computing, which is simple, practical and fast.
- Utilization of dataflow graphs for automatic test data generation.
- Accomplishment of successful testing coverage according to the All_DU_Paths data flow criterion, the level of which reaches 100% on the sample programs used for validation. This level of coverage outperforms other related research studies in this field.

The rest of the paper is organised as follows: Section 2 discusses briefly related work on the subject and section 3 describes the proposed framework. Section 4 presents the empirical results and, finally, section 5 concludes the paper and provides further research steps.

0-7695-2983-6/07 $25.00 © 2007 IEEE
DOI 10.1109/CIT.2007.97

2. Related research

Test data generation systems simulate the execution of a program to generate an adequate set of test cases with respect to a control flow coverage criterion, such as the statement, edge and decision [5, 6], or a data flow criterion, such as the all-nodes, all-edges and all-uses [7, 8]. The use of a control flow criterion implies analysis of the program's structure, e.g. branches and loops, whereas for a data flow criterion it is necessary to examine the data part of the program, e.g. variable to value bounding relation and variable usage within the program. Existing research addresses the challenges of defining and using robust data flow coverage criteria that could be comparable to those existing for control flow graphs, i.e. achieving similar or better results. The most relevant research studies are presented next.

Laski and Corel propose a strategy that uses the definition – *use* chain of a variable in order to guide the program testing [9]. This approach defines two different criteria, which are both based on the examination 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 [9], while improved definitions are given in [10].

Ntafos also proposed a method for selecting paths, namely *k-dr* interactions [11, 12]. 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) [12]. 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 [13]. 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, [14]. 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 [10, 13, 15]. In [15], Rapps and Weyuker 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 the criteria or revealing the weaknesses and strengths of each proposal. In [10], the authors claim and prove that after slight modifications on some criteria, the hierarchy can be formed as depicted in Figure 1. Comparison of the same criteria was also provided by Ntafos in [16].

The proposed testing approach can generate automatically test data to satisfy one of the top data flow criteria, namely the All-DU-Paths. 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.

3. The testing framework

The testing framework, which constitutes the core of this work, is divided into two parts, the Basic Program Analyzer System (BPAS) and the Automatic Test Cases Generation System (ATCGS). These two components analyse the code of a given program and determine the optimum set of test cases, respectively. The following sections describe in detail the aforementioned systems.

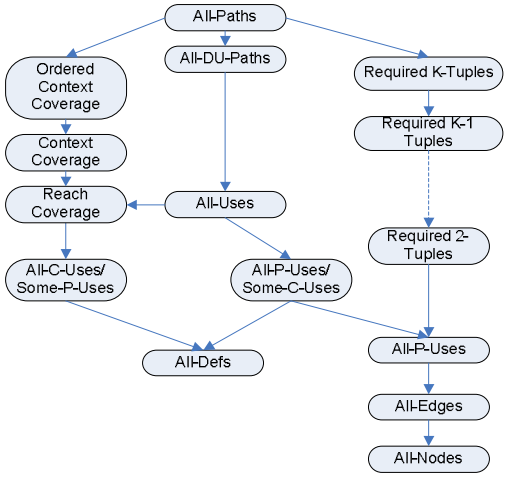

Figure 1: Hierarchy of data flow criteria [10]

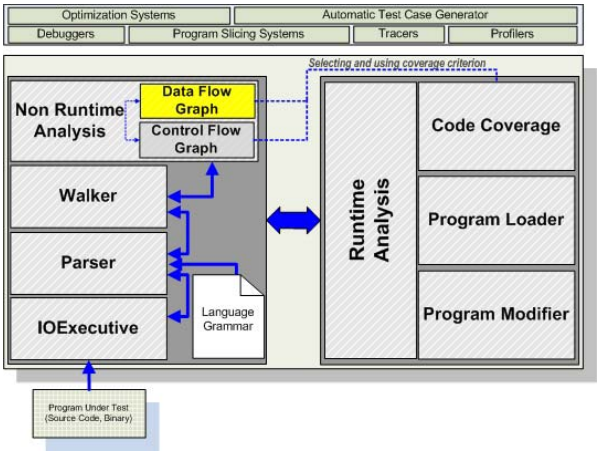

Figure 2: BPAS extended to support data flow graph

3.1 The Basic Program Analyzer System

The BPAS is divided into the runtime and non-runtime sub-systems, and it is capable of analysing large and complex programs [4]. Non-runtime analysis is carried out statically, i.e. without executing the program under study, while runtime analysis evaluates a program based on its behaviour during execution. The BPAS has been designed and built so as to provide, in the future, interfaces to other advanced application frameworks, e.g. Optimization, Test Data Generation, Program Slicing, etc. The key features of the BPAS are the extraction of essential program information and the creation of the corresponding control flow graph. In this work, the BPAS is extended to support the development of the data flow graph, which is built using the information provided by the corresponding control flow graph of a program under test. Figure 2 presents the architecture of the BPAS enhanced with the data flow module.

The Code Coverage module, which is part of the runtime analysis, is able to simulate the execution of the program under test for a given test case, and determine the part of the code executed. This module utilizes either the control flow or the data flow graph built earlier by the non-runtime module. Selecting the control flow graph implies that the testing coverage is a function of a selected control flow coverage criterion. Similarly, choosing execution based on the data flow graph results in the production of test data according to a specific a data flow criterion.

3.2 The Automatic Test Cases Generation System (ATCGS) and data flow criteria

The ATCGS utilizes the All-DU-Paths criterion, which is high in the hierarchy of the data flow criteria and hence it provides better results in terms of testing coverage. The operation of the ATCGS is based on genetic algorithms (GA) for searching the input space and determining the optimum set of test cases with respect to this data flow criterion.

GA constitutes a widely used optimization technique, which maintains and evolves a population of candidate solutions (called also individuals or chromosomes) to the problem at hand. The evolution takes place through generations by mimicking natural evolution, that is, through crossover and mutation of the current population. The fitness of each of each individual is assessed via a specific function which embeds the optimization criterion. In our case a new GA has been designed, each gene of which (i.e. part of the solution) encodes a data structure that encapsulates information for a distinct program input parameter. This structure contains its initial and current values of the parameter and its boundaries, i.e. maximum and minimum values. A chromosome (a set of genes), encodes the complete test case. For example, if a program has three inputs, say x, y and z, then each chromosome will have three genes, each encoding one of the parameters. Figure 3 shows the design of the chromosome for this example. Our encoding approach resembles the one reported in [6]; the authors of this study, though, evaluate their chromosomes using a distance fitness function only for the edge/condition coverage criterion (i.e. control flow criterion). In our case, a special-purpose fitness function is developed to

guide the algorithm effectively to the optimum set of test cases. The fitness function takes into account not only the number of the covered DU-Paths, but also how effective is a chromosome compared to the rest of the chromosomes in the current population and with respect to the set of paths at hand. The fitness function used is mathematically expressed in equation (1).

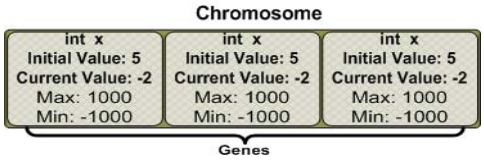

Figure 3: Chromosome encoding

$$F = \frac{Paths\,Covered}{Total\,\#of\,Paths} + \frac{Newly\,Covered\,Paths}{Paths\,Not\,Yet\,Covered} + \frac{Rarerly\,Covered\,Paths}{Paths\,re-Covered} \quad (1)$$

The number returned from the first part of eq.(1) is the most common in literature and shows the percentage of coverage for the gene considering the total number of paths that have to be covered according to the selected criterion. We should stress out at this point that using only this metric is not enough to evolve better genes in successive generations. Therefore, we resort to using two additional parts in eq.(1). The second and third parts of the fitness function may be considered a kind of reward to the chromosome. These are introduced based on the observation that a chromosome which only covers already covered paths (in a former or the current generation) is not really a step towards optimization. The second part of the fitness function gives the true contribution of the chromosome in terms of covering the targeted paths. The percentage of the paths covered for the first time by the chromosome under investigation is more important than the total number of paths covered. The final part of eq.(1) rewards chromosomes that achieve coverage of rarely covered paths. A path that is not covered by a lot of possible solutions might have a uniqueness requiring special conditions from the input variables. These conditions may be, for example, a special relation between the input variables (e.g. one of them has to be 0 and the other less than a specific value, or the first variable must be twice the value of the second one, etc.). Chromosomes that achieve coverage of those paths are rewarded.

Figure 4 presents a sample of the user interface that enables interaction with the program analysis and testing systems of the framework. The user may select a source file to be analysed by the program analysis module; based on the configuration settings, the system creates either its control flow graph, or its data flow graph, to be used by the testing system. Other system settings include the specification of the parameters boundaries, the mutation and crossover rate values and the selection operator, e.g. roulette wheel or tournament. The prototype application generates a set of test cases (see centre of figure) and allows the interaction with the active graph, the visualization of the results and the marking of the executed paths for each test case.

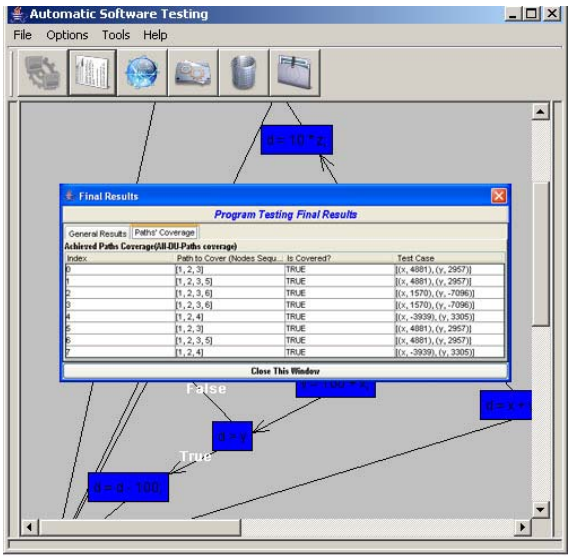

Figure 4: Snapshot of the prototype software system

The settings of the experimental results presented in the next section are as follows: Population was set to 50 individuals (chromosomes); the probability for both crossover and mutation ranges between 0.40 and 0.50 after empirical investigation that suggested that these values yield the best results; the algorithm was set to a maximum of 2000 generations (evolutions) with elitism in place (best chromosome of each generation passes to the next one). The user may choose to terminate the execution once the criteria are covered and thus stop the algorithm before the maximum number of evolutions is reached. This was actually the case in all the experiments included in this paper. Finally, the Roulette Wheel selection operator selected the chromosomes for the next generation.

4. Experimental Results

This section presents some preliminary results as the proposed testing framework is currently at an experimental level. The experiments were conducted

on a set of java programs listed in the first two columns of Table 1, which vary in size and complexity. The complexity of a program is defined as high (H) if one or more multiple conditions are present with more than two predicates, e.g. $A \vee B \wedge C$. If the program contains multiple conditions with maximum two predicates, e.g. $A \vee B$ or $A \wedge B$, then its complexity is characterized as medium (M), otherwise it is classified as simple (S). The columns of Table 1 describe the following: LOC are the lines of code, #TC the number of test cases produced, #Ifs the number of IF-statements present in the program, #NIfs the nesting level of the IF-statements, COM corresponds to the program's complexity, COV is the coverage percentage according to the selected criterion and TIME the time in seconds for execution of the test data generation algorithm.

Table 1: Experimental results on a set of sample programs ranging in size and complexity

ID	LOC	# TC	# *Ifs*	# NIFs	COM	COV	TIME (sec.)
1	20	2	1	0	S	100%	0
2	20	4	2	1	M	100%	0
3	20	4	3	1	H	100%	0
4	50	4	2	1	S	100%	1
5	50	8	4	1	M	100%	1
6	100	7	3	1	S	100%	15
7	100	7	4	2	S	100%	29
8	250	9	4	3	M	100%	37
9	250	10	5	2	M	100%	42
10	500	12	5	2	S	100%	49

It is clear from Table 1 that the proposed framework was able to generate test data automatically that cover successfully the sample programs under the All-DU-paths criterion, irrespectively of the number and nesting level of control statements or the type of complexity present in the code. It should be noted at this point that the results of Table 1 are the average values of 15 repetitions to account for the random and heuristic nature of genetic algorithms. Figure 5 depicts the number of test cases generated by the framework versus the lines of codes and complexity of the selected programs. As expected, the set of test cases increases in elements as the size of the programs increase. Figure 6 presents the execution time, which is the time required to generate the optimum set of test cases for achieving full coverage according to the All-DU-Paths criterion. As suggested by the figure, complexity plays less significant role compared to LOC as the time for simulating execution increases almost linearly with size.

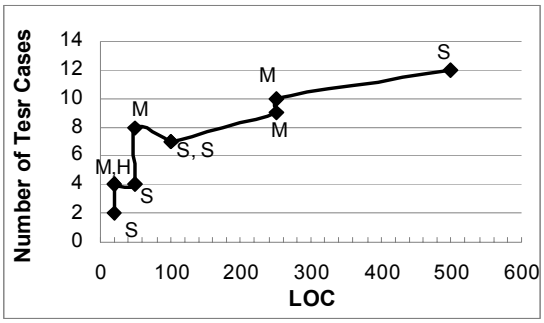

Figure 5: Number of test cases required for the All-DU-Paths criterion

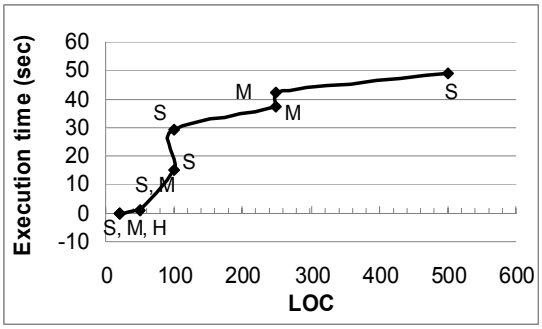

Figure 6: Time required for achieving 100% coverage against lines of code

Comparing our results to other data flow oriented test cases generator systems (e.g. [17]), we may conclude that our framework outperforms previous systems both in performance and functionality. For example, in terms of performance, the proposed framework not only is able to generate test cases for more complicate and larger programs than those reported in other studies, but it also achieves better testing coverage in shorter time.

5. Conclusions

This paper described an extension to a complete dynamic test cases generation framework. The framework comprises a program analyser and a test cases generator. The former parses a program and creates its control flow or data flow graph. The latter utilises genetic algorithms on the active flow graph to generate test cases that satisfy control flow or data flow testing criteria. The main contribution of this work is the integration of the data flow graph with the existing program analyser and the design of a special form of genetic algorithm for producing automatically test

cases with respect to a highly demanding data flow coverage criterion, namely the All_DU_Paths. Preliminary results showed that the proposed architecture is quite promising as it was able to automatically generate test cases for small to medium sized programs of different levels of complexity with 100% coverage. These results indicate better performance compared to similar studies, both in terms of testing adequacy and size/complexity of experimental sample code.

Future work will carry out further experiments with larger and more complex operating in realistic environments. Furthermore, the architecture will be extended to support features of object oriented programs, such as interfaces, inheritance, polymorphism and dynamic binding.

References

[1] P. McMinn, "Search-Based Software Test Data Generation: A Survey," Software Testing, Verification and Reliability, vol. 14, No. 2, pp. 105-156, 2004.

[2] G. M. Kapfhammer, "Software testing," in Tucker, A. ed., The Computer Science Handbook , Boca Raton, FL: CRC Press, 2004, chapter 105.

[3] A. A. Sofokleous, A. S. Andreou and G. Ioakim, "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, 2006, pp. 259-262.

[4] A. A. Sofokleous, A. S. Andreou, C. Schizas and G. Ioakim, "Extending and enhancing a basic program analysis system," in Proceedings of the IADIS International Conference, Applied Computing 2006 (AC2006), San Sebastian, Spain, 2006.

[5] B. Korel, "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, 1996, pp. 209-215.

[6] C. C. Michael, G. McGraw and M. A. Schatz, "Generating Software Test Data by Evolution," IEEE Transactions on Software Engineering, vol. 27, No. 12, pp. 1085-1110, 2001.

[7] J. Zhao, "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, 2003, pp. 188-197.

[8] J. H. Andrews, L. C. Briand, Y. Labiche and A. S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," IEEE Transactions on Software Engineering, vol. 32, No. 8, pp. 608-624, 2006.

[9] J. W. Laski and B. Korel, "Data flow oriented program testing strategy," IEEE Transactions on Software Engineering, vol. 9, No. 3, pp. 347-354, 1983.

[10] L. A. Clarke, A. Podgurski, D. J. Richardson and S. J. Zeil, "A Formal Evaluation of Data Flow Path Selection Criteria," IEEE Transactions on Software Engineering, vol. 15, No. 11, pp. 1318-1332, 1989.

[11] S. C. Ntafos, "On testing with required elements," in Proceedings of IEEE-CS COMPSAC, 1981, pp. 132-139.

[12] S. C. Ntafos, "On required element testing," IEEE Transactions on Software Engineering, vol. 10, No. 6, pp. 795-803, 1984.

[13] S. Rapps and E. J. Weyuker, "Data flow analysis techniques for test data selection," in Proceedings of the 6th IEEE-CS International Conference on Software Engineering, Tokyo, Japan, 1982, pp. 272-278.

[14] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," IEEE Transactions on Software Engineering, vol. 14, No. 10, pp. 1483-1498, 1988.

[15] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," IEEE Transactions on Software Engineering, vol. 11, No. 4, pp. 367-375, 1985.

[16] S. C. Ntafos, "A comparison of some structural testing strategies," IEEE Transactions on Software Engineering, vol. 14, No. 6, pp. 868-874, 1988.

[17] M. R. Girgis, "Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm," Journal of Universal Computer Science, vol. 11, No. 6, pp. 898-915, 2005.