AUTOMATIC SEARCH-BASED TESTING WITH THE REQUIRED K-TUPLES CRITERION

Anastasis A. Sofokleous, Andria Krokou
Department of Computer Science, University of Cyprus, Nicosia, Cyprus

Andreas S. Andreou
Department of Electrical Engineering and Information Technologies, Cyprus University of Technology
31 Archbishop Kyprianos Str. 3036, Lemesos, Cyprus

Keywords: Automatic Testing, Required k-tuples, Genetic Algorithms.

Abstract: This paper examines the use of data flow criteria in software testing and uses evolutionary algorithms to automate the generation of test data with respect to the required k-tuples criterion. The proposed approach is incorporated into an existing test data generation framework consisting of a program analyzer and a test data generator. The former analyses JAVA programs, creates control and data flow graphs, generates paths in relation to data flow dependencies, simulates test cases execution and determines code coverage on the control flow graphs. The test data generator takes advantage of the program analyzer capabilities and generates test cases by utilizing a series of genetic algorithms. The performance of the framework is compared to similar methods and evaluated using both standard and randomly generated JAVA programs. The preliminary results demonstrate the efficacy and efficiency of this approach.

1 INTRODUCTION

Software testing approaches that follow a structural (white-box) scheme, use the source code to reveal any errors, whereas in a functional (black box) scheme the testing process does not rely on the actual source code and the testing techniques use only the specifications of the program under testing; a combination of both schemes called gray-box has been also pursued (Nebut and Fleurey 2006). This work focuses on white-box testing using JAVA source code and automatically adapting the testing process based on the output of the test cases exercised on the code.

This work presents the implementation of the required k-tuples criterion, a data flow criterion incorporated into the automatic test data generation framework presented in (Csallner and Smaragdakis 2004). The framework utilizes a fusion of program analysis and test data generation techniques in order to parse the source code of the program under testing, create the control and data flow graphs, extract paths and determine a near to optimum set of test cases according to a coverage criterion (Korel 1996). This work extends the framework in order to support the required k-tuples criterion and uses a new path extraction algorithm, along with the dependencies defined on the data flow graphs, to generate the required k-tuples paths. A new approach to both the encoding of the chromosomes of the genetic algorithm and the way the fitness function is calculated allows the test data generator to achieve high coverage with respect to the required k-tuples criterion (Sofokleous and Andreou 2008b).

The rest of this paper is organized as follows: Section 2 presents related work on test data generation and briefly discusses similar approaches. Section 3 describes the proposed approach, while section 4 presents an assessment of its performance over a number of standard and sample programs and provides a short comparison with other approaches. Finally, the last section concludes the paper and outlines future research steps.

2 RELATED WORK

Test cases generation systems aim to determine an optimum set of test cases with respect to a testing coverage criterion (Frankl and Weyuker 1988). While there is a variety of testing criteria, the most widely known and pursued are the ones which are defined with respect to control and data flow graphs (Clarke *et al* 1989). Related research focused more on control flow criteria, such as the statement and edge, as for example the work of (Sofokleous and Andreou 2008b), which pursues high testing adequacy based on the edge/condition control flow criterion (Michael *et al* 2001, Pargas *et al* 1999, Harman 2007).. To achieve high coverage, this paper utilizes two algorithms, the first runs on the complete control flow graph to generate test data massively, whereas the second algorithm is executed on partial control flow graphs created dynamically according to uncovered paths.

Recent challenges involve the definition and implementation of robust data flow coverage criteria that could be equal or better than the control flow criteria. This paper presents an attempt to produce automatically test data with the required k-tuples data flow criterion (see section 3) (Ntafos 1981, Ntafos 1984) by utilizing evolutionary algorithms. To the best of our knowledge, implementation work and empirical results on this particular criterion has not been reported elsewhere, despite the fact that this criterion can achieve better results compared to those reported thus far. This work extends previous work that uses the ALL-DU Paths data flow criterion (Andreou *et al* 2007, Sofokleous and Andreou 2008a), where the execution of a test case is simulated with control flow graphs and the results of the execution are evaluated using data flow graphs. We should note here that the ALL-DU Paths criterion is a data flow criterion proposed by Rapps and Weyuker, 1982. While the ALL-DU Paths criterion has not been compared empirically with the criterion used in this work, many authors support that the two criteria are equal and can determine the same errors in a program under testing (Clarke *et al* 1989, Ntafos 1988). In section 4, however, we show that in some cases the required k-tuples criterion generates more paths than the ALL-DU Paths criterion, which implies a higher level of testing capability. Section 4 also compares the performance of our test data generator with a generator that uses the ALL-DU Paths criterion. Note that some of the original definitions of the data flow criteria are ambiguous and, in some cases, differ from the objective set by their authors (Clarke *et al* 1989).

Recent work on data flow-based testing can be found in (Ghiduk *et al* 2007), where the authors use genetic algorithms to generate test data according to data flow criteria. Their approach uses a multi-objective fitness function to evaluate the produced data and experiments have showed that their approach is more efficient over a random test data generator. A similar approach has been used to generate test data for FORTRAN programs (Girgis 2005). Data flow based test data generation has been also addressed in (Khamis *et al* 2000). This approach supports both arrays and loops, and the data generator domain uses partition and reduction methods on the input data in order to improve the performance of the generator.

The objective of this work is to present an evaluation of the *required k-tuples* criterion and implement a test data generator that can work efficiently based on this criterion. The genetic algorithm implemented for this purpose is guided by the data flow dependencies given as input to the test data generator.

The next section shows the implementation details of the framework supporting the criterion.

3 FRAMEWORK LAYOUT

This paper extends the Automatic Test Cases Generation System (ATCGS) presented in (Sofokleous and Andreou 2008b). ATCGS is a graphical user interactive system that analyses JAVA programs, creates control flow graphs, generates test cases and evaluates test data according to control flow criteria. An extended version of the framework was introduced in (Andreou *et al* 2007, Sofokleous and Andreou 2008a) that generate test data for data flow paths produced according to the ALL-USE data flow criterion.

The contributions of this work may be summarized as follows:

(i) ATCGS is the first tool reported in literature that can generate test cases according to the required k-tuples data flow criterion; to achieve the latter, the system has been enhanced with new modules and techniques. First, it creates data flow graphs and utilizes embedded data flow algorithms in order to generate the paths. Second, the system formulates test data generation as an optimization problem and utilizes specially designed genetic algorithms to solve it, the details of which are given in subsequent sections.

(iI) This is the first study reporting empirical results with this particular data flow criterion. Experiments thus far show the efficacy of our system and depict the applicability of this kind of testing. Both standard and sample JAVA programs used in this work are available for downloading from http:// www.cs.ucy.ac.cy/~asofok/testing/testdata.html.

3.1 The Required k-tuples Criterion in the Basic Program Analyzer System (BPAS)

In this work BPAS (Sofokleous and Andreou 2008b) is modified and extended so as to use the control flow graph and create its corresponding data flow graph, which is also presented graphically to the user. The data flow graph is used for generating the paths which will be executed through test cases provided by the test data generator. The total testing coverage is expressed in relation to the coverage percentage of these paths.

Compared to previous work (Sofokleous and Andreou 2008b), the test data generator is now able to generate test cases in relation to the *required k-tuples* criterion. According to this criterion the paths are propagated using the *k-dr interactions* method and the data flow graph of the program under testing (Ntafos 1981, Ntafos 1984). In such types of graphs, a variable can take any of the forms of a definition (**def**), or a computation (**c-use**) or a predicate (**p-use**) (Frankl and Weyuker 1988). Interactions between different variables are captured in terms of alternating definitions and uses, called *k-dr* interactions; an m-interaction is defined as $\lfloor d_{n_1}^{x_1}, u_{n_1}^{x_1}, d_{n_2}^{x_2}, u_{n_2}^{x_2}, d_{n_3}^{x_3}, u_{n_{13}}^{x_3}, \ldots, d_{n_m}^{x_m}, u_{n_m}^{x_m} \rfloor$, where variable x_1 is defined at node n_1 and used at node n_2, variable x_2 is defined at node n_2 and used at node n_3, etc. Based on number *k*, which is set by the user, BPAS generates all possible paths that satisfy the *k-dr interaction* criterion.

Figures 1a and 1b depict the nodes that satisfy the *1-dr* and *2-dr* interaction criteria, respectively. A **def-clear** path between two nodes, with respect to variable X, is a path on which none of its nodes is a definition (**def**) of X. Note that a *k-interaction* set of paths includes also all the *j-interaction* paths, where *j=1,...,k*.

If a graph contains one or more loop blocks, e.g. a representation of a *FOR* or *WHILE* loop, then, one or more *k-dr interaction* paths may be formed using a sequence of nodes from any of its possible executions, i.e. the loop can be traversed *n* times, where $n=0,1,\ldots,\infty$. For example, consider Figure 1c, which shows the source code of a loop. The problem in this case is that the particular code can populate an undefined number of *2-dr interaction* paths, the coverage of which cannot be guaranteed even by exhausting testing. For each loop, we only populate two groups of paths, if such paths exist; the first group describes *k-dr interaction* paths that can be populated while not entering the loop (i.e. 0-iteration of the loop), whereas the second group describes the *k-dr interaction* paths that can be populated by traversing *i=1,...,k* times the loop. Note that for the latter if at least one path cannot be found in a maximum of *k-iterations* of the loop, then a *k-dr interaction* path cannot exist even for the case where the loop is iterated *w-times*, where *w>k*.

3.2 Generating Test Data with ATCGS for the Required k-tuples Criterion

ATCGS communicate with BPAS to utilise the analysis modules of the latter and then searches the input space in order to determine and select a near to optimum set of test cases in relation to the *required k-tuples* criterion. The modified ATCGS follows a focus-based approach as opposed to the batch optimistic generation described in previous work. Basically, ATCGS utilises sequentially a series of genetic algorithms, one GA for each path; if $P=\{p_1, p_2, \ldots, p_k\}$ is the set of paths generated by BPAS, the invocation of the i^{th} GA focuses on path p_i, initially $i=1$. The basic steps of a standard genetic algorithm are adapted accordingly to reflect the problem addressed in this work as follows:

Initial Population Generation: First the GA generates the initial population. Each chromosome describes a test case, whereas each gene represents one of the input variables of the code under testing; some of the details encoded in a gene are the variable name, type and initial value. Suppose the testing code entails *x*, *y* and *z* input variables, then each chromosome will contain three genes to describe these three variables. Note that both the size of the chromosomes and the content of the genes depend on the program under testing and is automatically adapted.

Evaluation: The GA uses a specially designed fitness function (see section 3.2.1) which helps adapting its behaviour based on the selected path; the objective of the fitness function is to guide the search process to determine the test case that can cover the selected path p_i. The GA communicates with BPAS so as to execute the test case of each chromosome, identify the executed nodes and determine the coverage in relation to path p_i. If a test case that covers path p_i is found, then the GA terminates and the path is removed from the uncovered set of paths, i.e. $P'=P-\{p_i\}$; in this case, ATCGS proceeds to the next uncover path, if there is one, otherwise it presents the final results to the user. The GA may also terminate if it reaches a maximum number of generations defined by the user. Note that while the fitness value of each chromosome is calculated based on the selected path p_i, the GA may discover that the test case of a chromosome accidentally covers a different path, say p_j, where $j \neq i$ and $p_j \in P$; in this case, the GA associates the test case to the path that it accidentally covers, removes the path from the set P, and continues the search for path p_i.

Selection: The roulette wheel selection operator selects the chromosomes to participate to the next generation. Selected **chromosomes** are entered to a pool that is used in the next phase (Mitchell 1999).

Reproduction: Chromosomes are reproduced with crossover and mutation operations (Michalewicz 1996). The algorithm, then, proceeds to the evaluation phase.

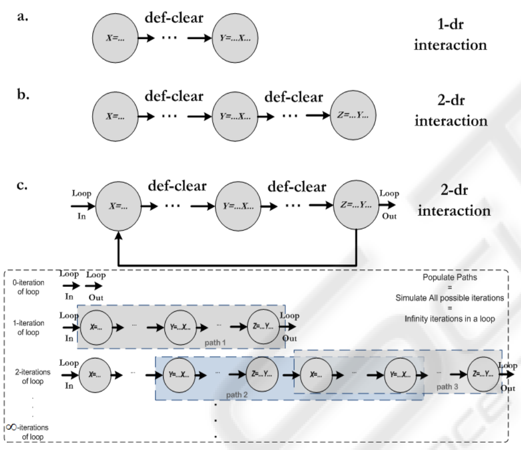

Figure 1: (a) 1-dr interaction, (b) 2-dr interaction, (c) populating 2-dr interaction paths for the loop.

3.3 The Fitness Function for the k-dr Interaction Paths

The fitness function is dynamic as it depends on the selected path; thus, the same chromosome may have two different fitness values if it is evaluated in relation to two different paths. The fitness function is expressed as follows:

$$F(p_i, TC_j) = \frac{p_i \text{ nodes covered using } TC_j}{\#\text{nodes in } P_i} \quad (1)$$

where p_i is the selected path and TC_j is the test case encoded in a chromosome of the population in the current generation. The maximum fitness value is 1, which denotes that test case TC_j achieves full coverage on path p_i. If the fitness function returns 1, then the current GA terminates and ATCGS selects the next path in the sequence. However, if the GA is unable to find a test case after the predefined maximum number of evolutions, then it assumes that this is a dead path and terminates the current process so as to allow ATCGS to continue with the rest of the uncovered paths.

4 EXPERIMENTAL RESULTS

This section describes several experiments carried out on both standard and sample JAVA programs. The experiments were executed on a computer with Intel Pentium 4 processor at 3.6GHz, 2GB memory Ram and JDK 1.6 running ON Windows XP Professional (SP2). The GA population size was set to 100 chromosomes, while the crossover and mutation rates were set to 0.5 and 0.2, respectively.

Section 4.1 presents the empirical results of the system over a series of standard programs and compares the proposed approach against previous work. Section 4.2 presents the results on a set of experiments carried out using a pool of sample programs with varying lines of codes (LOC) and complexity.

4.1 Experiments on Standard Programs

A number of standard programs have been selected as benchmarks. These programs are:

- **Fibonacci.java:** Returns the sum of *n* Fibonacci sequence of numbers.
- **FindMaximum.java:** Returns the largest between two numbers.
- **FindMinimum.java:** Returns the smallest between two numbers.
- **SumExample.java:** Returns the sum of *n* numbers, where *n* is given as a parameter.

Table 1 lists the results of the execution on the standard programs. The proposed system achieves 100% coverage with respect to the *1-dr* interaction criterion, whereas an algorithm that uses the *All-DU Paths* criterion achieves less coverage. The details of the *All-DU Paths* algorithm including the design details of the fitness function, can be found in (Andreou *et al* 2007). It is worth noting that in this set of experiments both criteria generate the same paths. The better performance exhibited by the proposed approach is due to the better design of the fitness function compared to (Andreou *et al* 2007): First it isolates each path, and therefore it does not bias the population with irrelevant information as it searches for a specific path; second, it guides the search process better as it provides an indication on how close the GA is to cover the selected path. Also, the good performance and efficiency of the proposed algorithm is obvious in the second series of experiments carried out using the *2-dr* interaction criterion; note that this set of experiments is feasible only in the present work.

Table 1: Comparative results using two different test data generation algorithms.

Program Name	**Required k-tuples Criterion (% coverage)**		**All-DU Paths (% coverage)** (Andreou *et al* 2007)
	1-dr	**2-dr**	
Fibonacci	100%	100%	83%
FindMaximum	100%	100%	100%
FindMinimum	100%	100%	100%
SumExample	100%	92%	91%

Table 2: Experiments: Complexity as (a) Low if the code does not contain nested IFs (b) Medium if it contains 1 nested IF, (c) High if it contains 2 or more nested IFs.

LOC	# nested if	# if	Complexity	#test cases	Coverage	Evolutions	Time (sec)
20	0	1	L	8	100 %	7	0
20	1	2	M	8	100 %	16	0
20	2	3	H	7	85 %	304	53
50	0	1	L	10	100 %	69	0
50	1	2	M	8	100 %	42	1
50	2	3	H	8	85 %	304	60
80	0	2	L	14	100 %	70	4
80	1	2	M	8	100 %	58	2
80	2	3	H	6	85%	303	60
100	0	4	L	14	100 %	50	17
100	1	3	M	13	100 %	58	5
100	2	4	H	14	85%	314	62

Both approaches of Table 1 extract paths according to their respective data criterion. If $P_{ALL\text{-}DU}$ and $P_{k\text{-}tuples}$ are the sets of paths extracted by the *ALL-DU Paths* and *required k-tuples* criteria, respectively, experiments show that $P_{ALL\text{-}DU} = P_{\text{k-tuples}}$, if *k=1*, whereas $\forall k > 1,\ P_{ALL\text{-}DU} \supseteq P_{k\text{-}tuples}$. The latter is supported by the fact that the paths are propagated according to *k-dr* interactions and involve the paths of all *j-dr* interactions, *j=1,…,k-1*; also, both of the criteria begin from a **definition** of a variable and then require the **use** of that variable (**p-use** or **c-use**), with the difference being, however, that in the *k-dr* interactions criterion, there are chains from **definitions** to **uses**.

4.2 Experiments on Sample Programs

Experiments reported in this section were carried out on a pool of programs varying on both their lines of codes (from 10 to 200 LOC) and their complexity (simple, medium, high). The programs were produced manually and do not serve any particular purpose. Note that testing is on a unit basis and LOC represent the size of a method; thus, testing larger programs is the same as aggregating the independent testing of many such methods. Furthermore, increasing the size (in terms of LOC) of a method does not affect complexity as the latter depends on the difficulty of covering a path, i.e. the condition that participates in a path.

The experimental results listed in Table 2 reveal a number of important conclusions. The more *IF* statements a program has, the more paths it contains and hence the more test cases are required to cover its paths; additionally, more time is required to determine the appropriate test cases for covering all these paths. As shown in the results, complexity plays a significant role along with LOC in the time required for executing the algorithms. The findings of Table 2 show also that in most cases the test data generator achieves a 100% testing coverage, while for programs with high complexity it manages to reach up to 85% testing coverage with respect to the *required k-tuples* criterion. The latter may be the result of many factors, such as the existence of dead code and the high complexity of the *IF* statements. For small and simple programs the system terminates in negligible time, whereas there is a linear dependency between time and lines of code. As the test data generator of this system works with a heuristic algorithm, each experimental result included in Table2 is the average value over 15 runs of the same experiment.

5 CONCLUSIONS AND FUTURE WORK

This paper presented the implementation of a testing approach based on *the required k-tuples* data flow criterion which depends on the paths propagated with k-interactions. The framework presented in this work uses a program analyser to generate the control and data flow graphs, builds dynamically the paths according to data flow criterion, and searches and discovers test cases for each of the paths; the user can view the test cases and interact with the graphs to view graphically the coverage of each test case. Experimental results were presented on a number of standard and random generated JAVA programs. Future work will carry out additional experiments using alternative implementations of the fitness function in order to compare their performance and investigate whether further improvements may be achieved. Additionally we plan to implement a representation model that will combine control flow graphs with UML diagrammatical notations. This will allow us to extend the representation models depicting the execution flow so as to support various features of object oriented code, such as interfaces aspects, inheritance, polymorphism and dynamic binding. Future work will also consider implementing other data flow criteria (Rapps and Weyuker 1982, Laski and Korel 1983), and compare them with the current and previous work. Our objective is to embed different types of errors in the programs and compare the efficiency of the implemented criteria in revealing these errors.

REFERENCES

Andreou, A. S., Economides, K. A. and Sofokleous, A. A., 2007, An automatic software test-data generation scheme based on data flow criteria and genetic algorithms, in: *Proceedings of the 7th IEEE International Conference on Computer and Information Technology,* Fukushima, Japan, October, (IEEE Computer Society: Los Alamitos, CA, USA), pp 867-872.

Clarke, L. A., Podgurski, A., Richardson, D. J. and Zeil, S. J., 1989, A Formal Evaluation of Data Flow Path Selection Criteria, *IEEE Transactions on Software Engineering,* 15(11), pp. 1318-1332.

Csallner, C. and Smaragdakis, Y., 2004, JCrasher: an automatic robustness tester for Java, *Software Practice and Experience,* 34(11), pp. 1025-1050.

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

Ghiduk, A. S., Harrold, M. J. and Girgis, M. R., 2007, Using Genetic Algorithms to Aid Test-Data Generation for Data-Flow Coverage, in: *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC '07),* Nagoya, Japan, December, (IEEE Computer Society: Washington, DC, USA), pp 41-48.

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

Harman, M., 2007, The Current State and Future of Search Based Software Engineering, in: *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007): Future of Software Engineering (FOSE '07),* Minneapolis, MN, USA, May 2007, (IEEE Computer Society: Los Alamitos, CA, USA), pp 342-357.

Khamis, A., Bahgar, R. and Abdelaziz, R., 2000, Automatic Test Data Generation Using Data Flow Information, *Dogus University Journal,* (2), pp. 140-153.

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, (ACM Press: pp 209-215.

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

Michael, C. C., Mcgraw, G. and Schatz, M. A., 2001, Generating Software Test Data by Evolution, *IEEE Transactions on Software Engineering,* 27(12), pp. 1085-1110.

Michalewicz, Z., 1996, *Genetic. Algorithms + Data Structures = Evolution Programs,* 3rd edn., (Springer-Verlag: London, UK).

Mitchell, M., 1999, *An Introduction to Genetic Algorithms,* 1st edn., (MIT Press: London, Uk).

Nebut, C. and Fleurey, F., 2006, Automatic Test Generation: A Use Case Driven Approach, *IEEE Transactions on Software Engineering,* 32(3), pp. 140-155.

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

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

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