A Symbolic Execution Tool Based on the Elimination of Infeasible Paths

Mike Papadakis and Nicos Malevris
Department of Informatics, Athens University of Economics and Business
Athens, Greece
{mpapad, ngm}@aueb.gr

Abstract - **Software testing forms a substantial activity of the software development cycle. Although important, it lacks from being automated mainly because of the various undecidable problems that it encounters. To this extend efficient heuristics have been proposed in order to bypass this problem. One such approach, called symbolic execution, is usually used for automating the test data generation activity. In this paper, an automated symbolic execution tool is proposed. The tool employs an efficient path heuristic, integrated with random testing for producing test cases. The tool handles the path explosion and constraint solving problems efficiently. This is achieved by targeting on specific likely to be feasible paths and by using a linear programming approach for the determination of their feasibility. Preliminary results are very encouraging as they show that a high coverage can be achieved within a limited amount of time-effort.**

Keywords - Testing Techniques, Symbolic Execution, Test Data Generation

I. INTRODUCTION

Software testing can be thought of as the process of determining the quality of a piece of software. This process requires a thorough examination of the code and the behavior of the software under test. Checking software's behavior requires its actual execution with test data. However, finding such input data forms a quite difficult and usually an impossible activity. Automating such an activity also faces many additional challenges such as pointer and array aliasing, solving of non linear constraint systems, the handling of loops, infeasible paths and the handling of multiple programming languages. To overcome these problems, many techniques, e.g., [1], [2], and tools, e.g., [3], [4] and [5], have been proposed in the literature. Although having been well studied, the problem remains open as most of the suggested techniques have been proven rather impractical in real situations. In the industry, the majority of the projects are carried out by generating test cases manually. Thus, providing an automated method/tool for producing the sought test data remains one of the main research issues of the software testing community.

Testing quality is usually evaluated based on various criteria known as adequacy criteria. These criteria help the tester to both guide and evaluate the test generation process. If for example, branch coverage were to be considered, the criterion guides the tester to produce test cases that cover yet uncovered branches. Additionally, it determines the level of testing thoroughness achieved by measuring the branch coverage level. All white box testing criteria serve these two above stated goals, by examining different test elements (i.e., statements, branches, data flow elements and linear code sequence and jump paths). Thus, by seeking the coverage of different test elements, testing results in diverse levels of test effectiveness. Additionally, different criteria also result in respective required effort levels in order to be fulfilled. This is a direct consequence of the number of test elements introduced by each criterion and the existence of infeasible paths. Infeasible paths have been identified as the most effort influencing factor of the testing activity [2], [6]. Moreover, a higher coverage level of 70% is often considered acceptable because of the effects of such paths [7].

This paper focuses on the efficient handling of infeasible paths in order to result into practical solutions for the software testing and maintenance faces. The basic idea behind the proposed approach is to perform symbolic execution [1] not to the entire program but only on a small set of paths, influenced by the considered test criterion. The selection of paths is performed based on the foundations of the Yates and Malevris method [2], a quite powerful method for alleviating the effects of infeasible paths. The undertaken approach, which is an extension of the above method uses a mixed symbolic and dynamic execution approach in order to effectively handle loops and difficult to reach program constructs. The suggestions of this paper have been incorporated into a unified symbolic execution prototype tool for the automated generation of branch tests.

The implemented prototype was built by extending and incorporating the experience gained from the development and the application of the Volcano tool [8]. The suggested prototype differs from Volcano in a number of issues. It uses a different path selection mechanism, which utilizes both symbolic execution and random testing. It also uses linear programming optimization methods for producing the test data. The paths selection and their symbolic evaluation is performed simultaneously rather than assigning a whole set of paths as input. The prototype proposed here, handles multiple programming languages by transforming them into the SymExLan scripting language [8]. This approach helps the prototype to effectively simplify the symbolic execution process and also to handle different programming languages in a straightforward fashion [8]. The prototype tool incorporates a verification mechanism in order to effectively combine static and dynamic analysis approaches such as symbolic execution, actual execution and random testing.

The primary aim of the developed tool was to produce an efficient and practical approach for the test data generation process. To this extent the chosen decision mechanism of the prototype was linear programming. This choice was motivated by the observation that the majority of time spent

978-0-7695-4144-0/10 $26.00 © 2010 IEEE
DOI 10.1109/ICSEA.2010.74

by the test data generation process is due to the constraint solving. Since linear optimization techniques have been identified as being quite efficient, their choice appears ideal for such a tool. The application of the prototype to a set of program units also signifies the tool's performance and the practicality of the proposed approach.

The rest of the paper is organized as follows: Section II introduces some relevant background material. Details of the underlying path selection method and its employment into the produced prototype are given in Sections III and IV respectively. Section V reports some results produced by the application of the prototype to a set of selected program units. Finally, Sections VI and VII discuss some relevant to the present work and the obtained conclusions.

II. Background

Automatically producing test inputs is usually performed by selecting a candidate set of paths based on which tests are drawn. This is achieved by employing either symbolic execution [1] or search based testing [9] approaches, guided by the selected paths. Symbolic execution [1] maps the program inputs and the paths computations onto a logical formula of constraints describing those computations. While the evolutionary based approaches use dynamic program executions and search based optimizations in order to drive the execution along the selected paths. This paper concentrates on the symbolic execution approach, but the incorporated path selection method may also be applicable to evolutionary based approaches.

The Control Flow Graph (CFG) of a program is a model of the program's flow represented by a graph. The CFG uses vertices to represent the program's basic blocks and arcs to indicate the flows between the basic blocks. A path is a finite sequence of vertices connected with arcs of the CFG. Programs usually contain an infinite number of paths, which in their majority are invalid as they do not form a valid program computation (no data can be found that execute them). These paths are called infeasible while the valid ones are called feasible. Thus, when employing a path based approach, the testing process can be regarded as the process of finding feasible paths, a tedious task especially if done in an ad hoc way [6].

Symbolic execution progresses by computing symbolic states along the selected program paths. A symbolic state is a mapping of the input program variables to symbolic values. Local variables are assigned their actual values based on the selected path or by the symbolic input ones. A set of constraints called path condition is constructed based on the symbolic states and the program decisions encountered through a selected path. The path condition constraints are called symbolic expressions and consist of symbolic variables or expressions over symbolic values. These expressions are computed by maintaining the symbolic states of a selected path and substituting the encountered values with those of the symbolic state. Solving the path conditions results in actual values forming program test inputs, which if input to the program under test, the selected path will be executed. If the path condition has no solution the path is infeasible.

Problems arise when the path conditions contain non linear expressions or pointer values. Non linear expressions need high computational resources in order to be solved or to be proven unsolvable. Similarly, pointer values require complicated and expensive analysis for tracking them precisely [10]. Furthermore, as most of the program paths are found to be infeasible i.e., [2], [6] there is a need to iteratively select and check alternative paths.

Branch testing based on selected paths entails the following steps:

1. Select a set of program paths that covers all the program branches.
2. Generate tests according to the selected set of paths.
3. Execute the program with the generated input data.

In the absence of infeasible paths any selected set of paths that cover all the program branches could be adequate for testing (performing steps 2 and 3 only once). In practice the existence of infeasible paths forces the process to iteratively perform steps 1 and 2 until the achieved coverage level reaches a predefined target.

From the above discussion it is evident that infeasible paths result in a waste of effort and thus, methods able to reduce their incidence are advisable. Such an approach is employed by the proposed tool and its details are given in the next section (Section III, path selection method).

III. Path Selection Method

In order to perform path testing efficiently, an appropriate path selection method is needed. In this section the foundations of such a method are given along with the details about its incorporation to the presented prototype tool implementation.

A. The extended shortest path method

The motivating idea behind the employed approach is to focus on a small set of specific but also likely to be feasible program paths. Unfortunately, this can only be performed heuristically [11]. Towards this direction, Yates and Hennell [12] advanced, and argued the following proposition:

A program path that involves $q >= 0$ predicates is more likely to be feasible than one involving $p > q$.

This proposition establishes the foundations for developing a heuristic approach that could serve as a yardstick for path testing. To this extend, Yates and Malevris [2] conducted a comprehensive study in order to statistically investigate its validity. The obtained results were in favor of the above proposition as they showed that with great statistical significance the feasibility of a path decays exponentially with the increasing number of predicates it involves. Based on this result, Yates and Malevris [2] proposed an extension to the Yates and Hennell [12] method by developing a path selection method with the ability to reduce in an a priori fashion the incidence of infeasible paths.

The method simply suggests that an advantage towards path feasibility can be achieved if paths that involve the minimum number of predicates are to be considered. Based on the above they suggested a path selection strategy based on the generation of k^{th}-shortest paths.

B. Integrating path selection and random testing

The Yates and Malevris [2] method originally proposed the consideration of the first k^{th}-shortest paths through a selected program branch. Here, this suggestion has been relaxed by considering only the k^{th}-shortest paths reaching a selected branch. By doing so, the considered paths involve a smaller number of predicates than with the complete paths, while maintaining the philosophy of the original one with respect to the number of predicates. By symbolically executing these paths, one can result in test inputs able to reach a target branch. This path is a sub path; the remainder of the path after the target branch up to the exit is determined dynamically in difference to the originally suggested method [2]. Here, data necessary for the program execution are derived based on the input values produced by symbolic execution together with random data produced for those inputs not considered symbolically. This consideration constitutes an amelioration of the initial method in [2] as it considers even shorter paths, hence providing more chances for these paths to be feasible while also eliminating the symbolic effort for their execution.

The utilized path selection strategy for purposes herein is detailed in Figure 1.

```
While (branch coverage < 1 AND $i \leq k$)
    Repeat steps 1, 2 and 3 once for each uncovered
    branch e in Branch Set
    1. Generate the next candidate path $\Pi_e^i$
    2. If path found to be feasible, then recalculate the
       value of coverage based on actual execution.
    3. Eliminate all covered elements from the Branch
       Set of uncovered branches
```

Figure 1. The utilized path selection strategy

Here, Π_e^i denotes the i^{th} shortest path from start to branch element e and k = maximum number of candidate paths (beyond which the generation of extra paths is prohibitive).

C. Incremental selection of paths (derivation of Π^i_e)

The employed strategy tries to overcome the difficulties imposed by non linear constraints and the existence of infeasible program paths by enforcing the selection of data along alternative paths. These paths are generated by selecting them as k^{th}-shortest paths, with respect to the number of predicates they contain. This set of paths consists of the shortest ones from the program's starting node to the target branching edge e. Hence, if $i < l \leq k$ then the length of path Π_e^i is less or equal to the length of path Π_e^l.

D. The criterion for selecting uncovered branches

In the case of uncovered branches additional paths are needed for which it is necessary to derive the test data. The order by which arcs should be selected is a question whose answer is derived again by using the philosophy that a program path with less predicates is more likely to be feasible than a rival containing more predicates. The policy adopted here is based on selecting the arc whose shortest path to reach it, is smaller in length than the other candidate shortest paths that reach the rest of the remaining arcs. In case of ties they are broken arbitrarily.

IV. Tool Description

This section gives details about the proposed prototype implementation. The tool generates automatically and executes test cases based on the suggestions made in the above sections. The prototype has been entirely built using the java programming language and by incorporating the LpSolve [13] linear programming package as a decision procedure. The implementation is composed of four modules detailed in the following subsections. An overview of the tool architecture is given in Figure 2.

A. Symexlan

The proposed prototype employs the SymExLan script language [8] based on which it performs the symbolic execution process. SymExLan is a language that acts as an intermediate representation of the structured programming languages such as C, Delphi etc. It also simplifies the symbolic execution process [8] as it simplifies the encountered expressions i.e., all compound predicates are transformed into multiple simple ones; it embodies all the necessary features for performing symbolic execution, such as the program CFG, program branches, preconditions, postconditions, e.t.c.. Additionally, any procedural programming language can be translated to this intermediate representation in order to advance the testing process. This activity in the present system is performed by the source to source compiler modules (Figure 2) that read the source code (SRC code) of the program under test and translate it to SymExLan script [8].

B. Symbolic Executor and Path Generator

The heart of the proposed system is the Symbolic executor and the path generation modules. These two modules are integrated in order to produce the required test cases. The former is responsible for generating the set of k^{th}-shortest paths, while the later is responsible for producing a linear programming problem according to each selected path. The transformation into a linear programming problem is straightforward as all produced constraints are simple ones. The objective function can be any trivial one, as any solution satisfying the path condition will be acceptable. It should be noted that logical expressions (e.g., logical AND, OR) are treated in a path based approach, as this is forced by the short circuit mechanism of modern programming languages [14].

One general problem encountered with constraint solving techniques is the handling of non-linear constraints. This problem is currently tackled indirectly based on the actual program execution. In future it is planed to incorporate linear relaxations as suggested in [15] and to incorporate a mixed symbolic and evolutionary based approach [16].

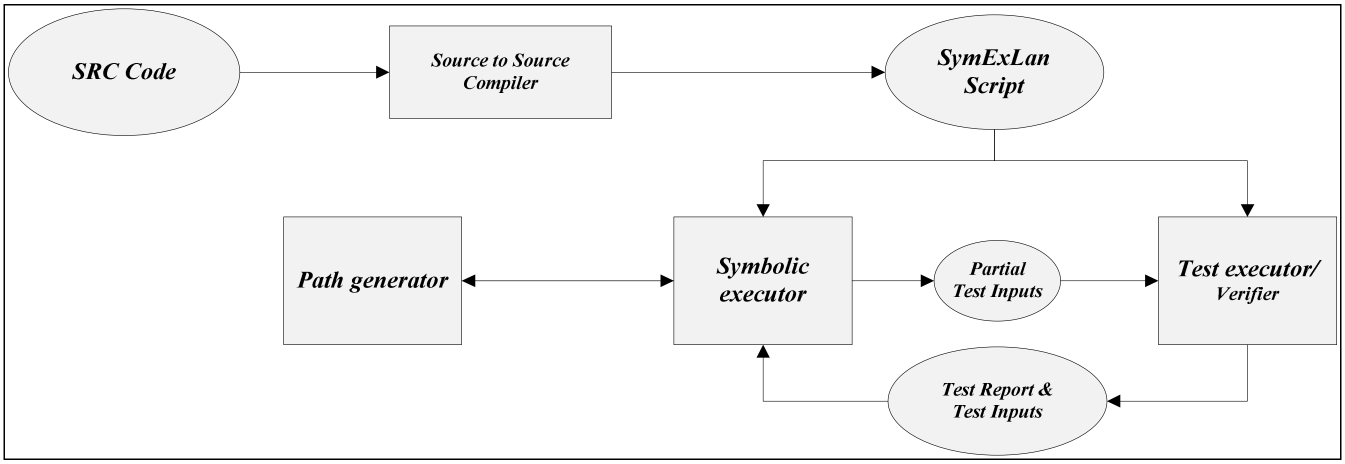

Figure 2. The prototype architecture

C. Test Executor/Verifier

The prototype combines symbolic execution and actual program execution approaches in order to be more efficient and effective. To achieve this, a simulator able to execute the test program execution is also required as this is now in SymExLan. This purpose is fulfilled by the test executor/verifier module. Its tasks are the determination of the complete execution program paths, the augmentation of the partial program inputs, if needed, with randomly generated ones (as described in Section III.B) and the verification of the entire process. This module actually forms an interpreter of the SymExLan script programs.

V. Experimental Results

This section reports results from the application of the proposed prototype. The evaluation is performed based on a sample of 20 program units consisting of 10 programs written in Delphi and 10 written in C/C++ programming languages used in previous studies in the literature. Table I presents the selected sample profile that includes the lines of code for the original program versions, the number of branches and the number of the infeasible branches that they contain. Here, it must be noted that the infeasible branches were excluded from the calculation of all the reported coverage scores.

The experimental evaluation was performed by considering three main points. First, a random test generation process was performed in order to provide a baseline about the tool's performance. Second, a test generation process was performed by setting an upper bound value for *k* equal to 50. This approach was originally suggested in [2] for providing a good cost effective heuristic approach to the testing process. Finally, the tool was employed by incorporating a relatively high *k* value, equal to 10,000.

All the selected test subjects were first transformed into the SymExLan script language. Using the produced scripts, a random generation of inputs was performed by selecting 1,000 tests. The achieved branch coverage of the random tests was computed by using the test executor/Verifier module in order to avoid any side effects of the translation. The achieved coverage is recorded in the column Random testing coverage of Table I. These results are given in order to highlight the effectiveness of the proposed approach. Random testing as can be seen in Table I achieves on average 60% of branch coverage, which is rather poor. Also, the achieved low scores of random testing, as recorded in Table I, indicate that (as discussed below) it is the combination of both symbolic execution and random testing that can achieve higher coverage and not random testing on its own.

The experiment is continued by employing the prototype for performing branch testing on the selected test subjects. The obtained results are presented in Tables II and III. Tables II and III record the achieved coverage (column Coverage), the total number of the examined paths (column Paths), the number of the produced test cases and the total time required by the employed tool, measured in seconds, (columns Tests and Sym. Ex. Time respectively).

Table II records the results from the application of the prototype in a cost effective fashion. Thus, a value of *k* = 50 was adopted. This *k*-value was found by the present experiment

TABLE I. Subject Programs

Test Subject	Lines of Code	No. of Branches	Random testing coverage	Infeasible Branches
D1	*52*	*34*	*47%*	*1*
D2	*40*	*31*	*55%*	*0*
D3	*66*	*40*	*33%*	*0*
D4	*36*	*21*	*100%*	*0*
D5	*56*	*27*	*89%*	*0*
D6	*40*	*31*	*61%*	*0*
D7	*45*	*33*	*88%*	*0*
D8	*50*	*26*	*50%*	*0*
D9	*48*	*56*	*82%*	*0*
D10	*75*	*48*	*30%*	*1*
C1	*60*	*51*	*43%*	*0*
C2	*70*	*62*	*10%*	*0*
C3	*35*	*27*	*59%*	*0*
C4	*30*	*25*	*76%*	*0*
C5	*26*	*18*	*66%*	*0*
C6	*40*	*32*	*81%*	*0*
C7	*30*	*31*	*74%*	*0*
C8	*26*	*28*	*46%*	*0*
C9	*20*	*33*	*58%*	*0*
C10	*26*	*25*	*36%*	*0*
Avearage	*44*	*34*	*60%*	*-*

to be the most appropriate one. It is also along the same lines with the one suggested in [2] giving a high confidence about its validity. From these results it is obvious that the proposed approach is quite effective and efficient as it achieves approximately 93% coverage in less than 1.5 second by examining 201 paths on average. Additionally, the worst case is 73%, nevertheless usually considered as an acceptable coverage level [7]. This is due to the ability of the employed path selection method to produce feasible paths. Table III records the results from the application of the prototype with its *k* parameter value set to 10,000. From these results it is evident that an additional coverage of 5% can be achieved by spending a considerable amount of time or effort compared to the one spent considering the cost effective application. Thus, this coverage raise is due to the examination of 5,137 paths on average, per program. The above results are statistically significant as derived by a *t – student* test performed on the coverage values of all three tables.

In order to present the abilities of the proposed tool to efficiently produce test cases it was also utilized on the Tcas program [17], a large (containing 173 lines of code) and also widely used in the literature program. The prototype required approximately 12 seconds to produce 16 branch adequate test cases. Although five program branches were left uncovered, a manual analysis showed that these were infeasible and thus the produced tests were branch adequate ones establishing the maximum possible coverage of 90%. It must be noted that the reported time is wall clock time that includes the tool's initialization, program parsing, test production and test simulated execution.

Conclusively, the random generation of test cases achieved on average a 60% of coverage while the employed tool achieved 93% of coverage in a cost effective-fashion by spending in total less than 25 seconds for all program units. An additional 5% of coverage can also be achieved with a surplus of approximately 6.5 minutes in total.

TABLE II. OBTAINED RESULTS FOR $K = 50$

Subject	Coverage	Paths	Tests	Sym. Ex. Time
D1	*94%*	*384*	*6*	*0.844091358*
D2	*74%*	*446*	*3*	*1.7888585*
D3	*100%*	*118*	*8*	*1.403037886*
D4	*100%*	*15*	*2*	*0.459138243*
D5	*89%*	*438*	*3*	*0.496860076*
D6	*100%*	*3*	*3*	*0.457364554*
D7	*73%*	*452*	*2*	*0.515380097*
D8	*100%*	*7*	*7*	*0.92713208*
D9	*93%*	*426*	*7*	*1.358667861*
D10	*91%*	*351*	*8*	*1.134668722*
C1	*100%*	*88*	*14*	*1.743730139*
C2	*90%*	*341*	*11*	*5.347082311*
C3	*93%*	*141*	*5*	*0.993307599*
C4	*76%*	*334*	*4*	*0.955805347*
C5	*100%*	*7*	*6*	*0.246913504*
C6	*94%*	*143*	*5*	*0.645474844*
C7	*97%*	*174*	*4*	*0.684474214*
C8	*100%*	*29*	*8*	*0.461636046*
C9	*100%*	*14*	*14*	*1.599474305*
C10	*92%*	*114*	*4*	*1.103021543*
Average	*93%*	*201*	*6*	*1.158305961*

TABLE III. OBTAINED RESULTS FOR $K = 10{,}000$

Subject	Coverage	Paths	Tests	Sym. Ex. Time
D1	*100%*	*887*	*7*	*1.722841172*
D2	*100%*	*1282*	*5*	*1.585620589*
D3	*100%*	*10068*	*8*	*59.0570393*
D4	*100%*	*15*	*2*	*0.459138243*
D5	*100%*	*6914*	*5*	*24.59993323*
D6	*100%*	*3*	*3*	*0.457364554*
D7	*94%*	*10837*	*7*	*7.183346277*
D8	*100%*	*7*	*7*	*0.92713208*
D9	*100%*	*624*	*9*	*1.205022636*
D10	*96%*	*10938*	*9*	*20.40800318*
C1	*100%*	*88*	*14*	*1.743730139*
C2	*100%*	*719*	*14*	*18.25030179*
C3	*100%*	*345*	*6*	*1.585201401*
C4	*76%*	*30186*	*5*	*122.3285885*
C5	*100%*	*7*	*6*	*0.246913504*
C6	*94%*	*10092*	*4*	*12.8882029*
C7	*97%*	*10130*	*5*	*17.45165342*
C8	*100%*	*29*	*8*	*0.461636046*
C9	*100%*	*14*	*14*	*1.599474305*
C10	*100%*	*9560*	*5*	*92.69842082*
Average	*98%*	*5137*	*7*	*19.3429782*

VI. RELATED WORK

A significant number of automated test data generation approaches have appeared in the literature. From the earliest approaches a well referenced one is that by Coward and Ince [18] who described the implementation of the SYM-BOL tool. This tool had several limitations such as the handling of arrays, pointers, infeasible paths and constraint solving. The suggestion made by Coward and Ince [18] was then incorporated in the Volcano tool [8] a prior work, which formed the basis for the presently suggested one. Volcano introduced the SymExLan script language in order to simplify the symbolic execution process and also made easy the handling of various programming languages. Additionally, Volcano proposed a possible solution to the array and pointer aliasing problems through the use of a memory table.

Another related to the present attempt is the one due to Lapierre et al. [19], who suggested using mixed integer linear programming and execution trees in order to automatically produce test data. This attempt is one of the first approaches that used linear optimization as a decision mechanism for the testing purposes. The present work differs from the above mentioned one in the way it handles and selects the candidate paths. In the Lapierre et al. [19] approach all paths contained in the symbolic execution tree up to a certain depth are considered. This leads to the consideration of many irrelevant up to the target branch paths resulting in an excessive amount of effort involved for their evaluation. On the contrary, the present approach takes advantage of a user defined *k* path generation parameter, resulting in cost effective solutions.

Recently, various tools have also been proposed. The symbolic extension of Java Path Finder (JPF) [5] is a tool that performs symbolic execution on java programs based on a model checker. Although being powerful it performs symbolic evaluation in an exhaustive manner, facing thus path exposition problems. Another tool named Euclide [3] has also been proposed for the verification of C programs. Euclide utilizes a constraint based approach in order to address the test data generation, the counter example generation and the partial program proving applications. Its application effectiveness relies on the incorporated constraint based solver rather than onto a path selection technique. PathCrawler [4] and Dart [20] and OSMOSE [21] are three path oriented tools utilizing actual and symbolic execution simultaneously. These tools select paths in an indirect way and usually consider all program paths, thus resulting in similar to the previously mentioned path explosion problems. Despite this, these approaches have the advantage of handling efficiently non linear and pointer constraints. Finally, OSMOSE has the advantage of working directly on machine executable code rather than on the program source code, handling effectively low-level program features [21].

VII. CONCLUSION

In this paper, an automated test data generation tool was proposed. The proposed tool utilizes an efficient path selection strategy integrated with random testing approaches in order to achieve a considerably high percentage of coverage within a limited amount of time. The innovative part of the proposed tool is its ability to produce effectively test cases by considering only a small part of the whole program path set. Additionally, as the proposed tool considers a different branch per turn it spends equal effort on all uncovered program branches at each iteration, avoiding being trapped on infeasible program test elements. The incremental in nature mechanism of the prototype guarantees that if the process stops at any stage, this should not result in favor of exercising some program points while not doing so with others. Thus, the process can be stopped at any point.

The undertaken experiments showed that a remarkable coverage achievement of 93% can be attained by employing only 50 program paths per branch. This issue signifies the ability of the proposed approach to produce feasible program paths. Such an effective heuristic could also be used in other software testing contexts in order to produce fast and in a large scale results. The prototype is quite flexible as it can be parameterized to achieve higher coverage in additional time. Thus, it is able to continue its process in order to attain higher levels of coverage as it was demonstrated on the selected sample of programs, which managed to achieve a 98% of branch testing.

Approaches similar to the present one could also be used in order to test software changes and thus perform regression testing. To achieve this efficiently, one could select *k*-shortest paths passing through some program changed points and produce the sought test data. Here, it should be mentioned that the employed approach could also reuse the path selection analysis of the unchanged program parts performed during the testing phase, resulting in additional time and effort savings at the regression testing process.

Future work is directed towards additional path heuristic approaches able to make the approach quite scalable, beyond the unit level. The integration of these heuristics with simple to detect infeasible path patterns as proposed in [22] is also under investigation. Finally, the extension of the present tool to include higher level testing criteria such as mutation testing [23] is also planned.

ACKNOWLEDGMENT

This work is supported by the Basic Research Funding (PEVE 2010) program of the Athens University of Economics and Business.

REFERENCES

[1] J.C. King, "Symbolic execution and program testing", Comm. ACM vol. 19, pp. 38-94, 1976.

[2] D. F. Yates, N. Malevris, "Reducing the effects of infeasible paths in branch testing", Software Engineering Notes, vol. 14, pp. 48-54, 1989.

[3] A. Gotlieb, "Euclide: A Constraint-Based Testing Framework for Critical C Programs". In ICST, 151-160, 2009.

[4] N. Williams, B. Marre, P. Mouy and M. Roger, "PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis", In EDCC, 281-292, 2005.

[5] W. Visser, C. Pasareanu and S. Khurshid, "Test input generation with java PathFinder". In ISSTA, 97-107, 2004.

[6] D. F. Yates and N. Malevris, "An objective comparison of the cost effectiveness of three testing methods", J. of Information & Software Technology, vol. 49, 2007, pp. 1045-1060.

[7] A. Von Mayrhauser. "Software testing: opportunity and nightmare", IEEE Int. Test Conf., pp. 551-552, 1992.

[8] C. Koutsikas and N. Malevris: A Unified Symbolic Execution System. In AICCSA, 466-469, 2001.

[9] B. Korel, "Automated software test data generation", IEEE Transactions on Software Engineering, 16(8):870-879, 1990.

[10] Nikolai Kosmatov, "All-Paths Test Generation for Programs with Internal Aliases", In ISSRE,147-156, 2008.

[11] H. N. Gabow, S. N. Maheshwari, and L. J. Osterweil, "On two problems in the generation of program test paths", IEEE Trans. on Softw. Eng., vol. SE-2, pp. 227-231, 1976.

[12] D. F. Yates and M. A. Hennell, "An approach to branch testing" In: 11th Workshop on Graph Theoretic Techniques in Computer Science, Wurtzburg, West Germany, pp. 42-433, 1985.

[13] lp_solve, http://lpsolve.sourceforge.net/, May 2010.

[14] M. R. Woodward and M. A. Hennell, "On the relationship between two control-flow coverage criteria: all JJ-paths and MCDC", Information & Software Technology 48(7), 433-440, 2006.

[15] T. Denmat, A. Gotlieb, and M. Ducasse. Improving constraint-based testing with dynamic linear relaxations. In ISSRE, 181-190, 2007.

[16] M. Papadakis and N. Malevris, "Improving Evolutionary Test Data Generation with the Aid of Symbolic Execution", AISEW, 201-210, 2009.

[17] J. Harrold and G. Rothermel. Siemens programs, HR variants. http://www.cc.gatech.edu/aristotle/Tools/subjects/, May 2010.

[18] P.D. Coward and D. Ince, "The symbolic execution of software: The SYM-BOL system", 1995, London: Chapman & Hall.

[19] S. Lapierre, E. Merlo, G. Savard, G. Antoniol, R. Fiutem and P. Tonella, "Automatic Unit Test Data Generation Using Mixed-Integer Linear Programming and Execution Trees", 189-198, ICSM 1999.

[20] P. Godefroid, N. Klarlund and K. Sen, "DART: directed automated random testing". *SIGPLAN Not.* 40(6), pp. 213-223. 2005.

[21] S. Bardin and P. Herrmann. "OSMOSE: Automatic Structural Testing of Executables". Software Testing, Verification and Reliability. To appear.

[22] M. N. Ngo and H. B. K. Tan, "Heuristics-based infeasible path detection for dynamic test data generation", Information & Software Technology 50(7-8), pp. 641-655, 2008

[23] M. Papadakis and N. Malevris, "An Effective Path Selection Strategy for Mutation Testing", Proc. APSEC, 2009.