SYMBOLIC EXECUTION FOR DYNAMIC, EVOLUTIONARY TEST DATA GENERATION

Anastasis A. Sofokleous, Andreas S. Andreou and Antonis Kourras
Department of Computer Science, University of Cyprus
75 Kallipoleos Str., P.O. Box 20537, CY1678, Nicosia, Cyprus

Keywords: Search-based test data generation, Symbolic program transformation, Genetic algorithms, Control flow graphs.

Abstract: This paper combines the advantages of symbolic execution with search based testing to produce automatically test data for JAVA programs. A framework is proposed comprising two systems which collaborate to generate test data. The first system is a program analyser capable of performing dynamic and static program analysis. The program analyser creates the control flow graph of the source code under testing and uses a symbolic transformation to simplify the graph and generate paths as independent control flow graphs. The second system is a test data generator that aims to create a set of test cases for covering each path. The implementation details of the framework, as well as the relevant experiments carried out on a number of JAVA programs are presented. The experimental results demonstrate the efficiency and efficacy of the framework and show that it can outperform the performance of related approaches.

1 INTRODUCTION

Many researchers have used control flow graphs to examine features of software and combine program analysis with other techniques such as the testing, slicing and optimisation. Recent research focuses on test data generators, systems that can generate test cases in relation to a testing coverage criterion. Most of the test data generators are either random (P. Godefroid, Klarlund, & Sen, 2005), if they generate test cases randomly, or dynamic, if they adapt their behaviour based on the generated data (Bertolino, 2007). Several authors use optimisation algorithms to guide the search process as the problem of generating test data is formulated as an optimisation problem (Pargas, Harrold, & Peck, 1999).

The problem, however, is that in some cases even optimisation algorithms may not be able to generate an adequate set of test cases with respect to the selected coverage criterion. This may be the result of the program complexity; for example executing a path may be more complicated if the path contains quite a few multiple conditions and hence the optimisation algorithm cannot achieve the desired value in each condition.

This paper addresses the complexity challenge of programs and aims to develop an efficient algorithm that can simplify this complexity and work together with a test data generator to produce the target set of test cases. We present the design and implementation details of a framework that utilises symbolic execution with evolutionary algorithms to generate test cases for JAVA programs. The symbolic execution, which is embedded in the program analyser of the framework, transforms the original program to a set of simple paths, the individual testing of which is equivalent to the testing of the original program. The paper also presents a set of experiments that demonstrate the successful performance of the framework in terms of coverage adequacy. The framework is compared with a similar method found in the relevant literature and the results show that the framework can achieve better coverage with respect to the criterion selected.

The rest of the paper is organized as follows: Section 2 presents some related work on this subject, while section 3 describes the proposed testing framework. Section 4 evaluates the efficacy of our testing approach and provides experimental results on a number of sample programs, as well as some commonly known programs that are used as benchmarks for comparison purposes. Finally, Section 5 concludes the paper and suggests future research steps.

Sofokleous A., Andreou A. and Kourras A. (2009).
SYMBOLIC EXECUTION FOR DYNAMIC, EVOLUTIONARY TEST DATA GENERATION.
In *Proceedings of the 11th International Conference on Enterprise Information Systems - Databases and Information Systems Integration*, pages 144-150
DOI: 10.5220/0001992701440150
Copyright © SciTePress

2 RELATED WORK

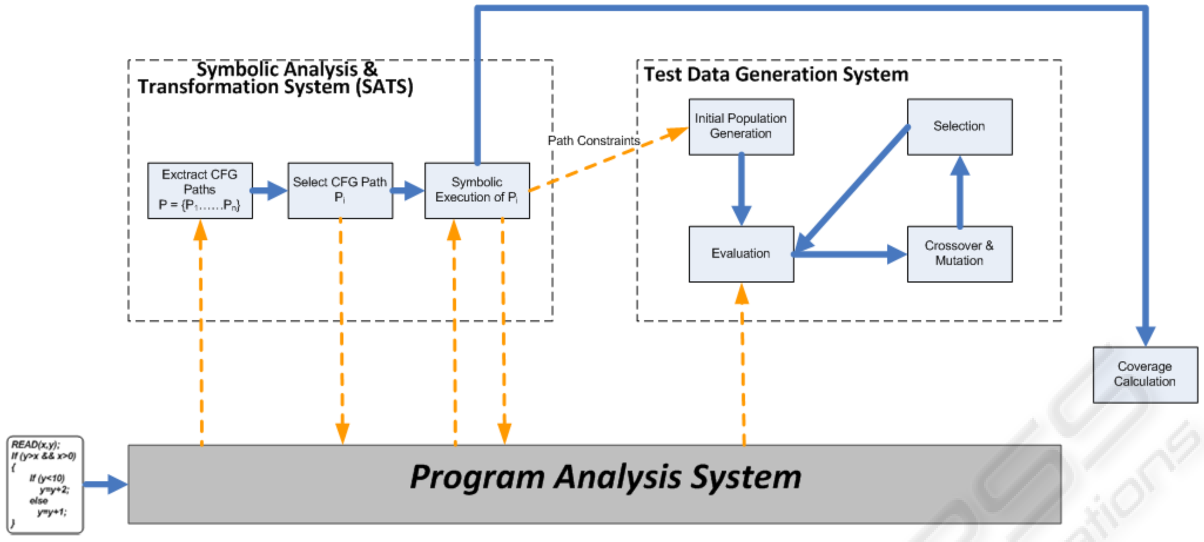
This paper proposes a hybrid software test data generation algorithm that utilizes symbolic execution to optimize (lower) the complexity of the programs under testing, and computational intelligent algorithms to generate test cases. Most test data generation approaches use the source code to analyze the program and guide the test process (Bertolino, 2007; McMinn, 2004; A. Sofokleous & Andreou, 2008).

Test data search can be utilized either on the complete program or according to a single path (A. Sofokleous & Andreou, 2007; Zhang, Xu, & Wang, 2004). Generated data are usually evaluated according to testing criteria varying on the implementation complexity and the testing quality offered (Frankl & Weyuker, 1988); examples of the branch coverage criterion and the branch/condition coverage criterion are shown in (Soffa, Mathur, & Gupta, 2000), (A. A. Sofokleous & Andreou, 2008), respectively. Genetic Algorithms (GA) have been used with remarkable success in dynamic test data generation as they can efficiently search the huge input space and determine test cases for complicate programs (P. Godefroid, 2007). Part of GA's success in this particular problem is the design of the fitness function; poor design of this function may misguide the search process and lead to over processing or even to the fitness landscape. The fitness landscape, which is one of the recent problems engaging researchers in this area, is a state of the GA where its fitness function gives the same value for almost all solutions; as a result, the search process cannot be guided to the right direction. Suppose the fitness function is designed to capture the distance from a search target, the fitness landscape may be caused in cases where more than one path can lead to a search target (i.e. path problem) or when one or more conditions take values from a small set of values (i.e. flag problem) (Baresel & Sthamer, 2003). To address the path problem, researchers in (McMinn, Harman, Binkley & Tonella, 2006) suggest generating test data for each path leading to its target, whereas to deal with the flag problem (Bottaci, 2002) a common way is to transform the program to multiple sub-programs that could maintain the same properties as the original version (Baresel, Binkley, Harman & Korel, 2004); in this case, generating test cases for each individual sub-program is the same as generating test cases for the original program. This paper uses both path isolation and symbolic program transformation to address efficiently both types of origins of the fitness landscape respectively. The advantage of our approach is that it addresses both the path and the flag problems using a novel, flexible method that combines symbolic execution with genetic algorithms.

Symbolic testing was first reported by King et al. back in 1976 (King, 1976). The need for symbolic execution comes as a consequence of the increase of software complexity. Symbolic testing is an abstract definition which has two implementation methods. The first is Symbolic Execution and the second is Symbolic Evaluation. An example of the former, along with extended finite state machines, is used in (Zhang et al., 2004) to capture the program's behaviour, extract feasible paths and generate test data for the program under testing. With the use of a control flow graph and symbolic execution, our approach extracts the paths of the program under testing and transforms each path to a set of equations. Each set of equations describes a set of conditions, the satisfaction of which implies the execution of the path. The combination of symbolic execution and control flow analysis has been also reported in (Kebbal, 2006), with some limitations, however, in the analysis of the graph which represents many statements in each block. The main problem with approaches following only symbolic testing is located in the loops; according to (Tillmann & Schulte, 2006), symbolic testing fails to reveal the needed test cases for executing a loop for a particular number of times. In our case, the equations describe the conditions that when satisfied can force such an execution for a specific loop, i.e. how to iterate *k* times the particular loop.

3 SYMBOLIC TRANSFORMATION AND EVOLUTIONARY TEST DATA GENERATION

This paper extends the Dynamic Test Data Generation Framework (DTDGF) described in (A. A. Sofokleous & Andreou, 2008). The DTDGF consists of two main systems, the Analysis and Testing systems. The former analyses programs, creates program representations such as control and data flow graphs, and simulates and reports the execution of a test case on the control flow graph. The Analysis System (AS) may be used by other systems performing testing, debugging, optimization and slicing. Currently, AS is integrated with a test data generator, a system that utilizes genetic



```
READ(x,y);
If (y>x && x>0)
{
    If (y<10)
       y=y+2;
    else
       y=y+1;
}
```

Figure 1: The Symbolic Analysis & Transformation System between the Program Analysis System and the Test Data Generation System.

algorithms to produce test cases in relation to control or data flow criteria.

The system presented in this paper uses the control flow graphs produced by AS to transform paths to sets of symbolic equations. A new module, called Symbolic Analysis & Transformation System (SATS), works between the AS and a revised version of the Test Data Generation System (Figure 1).

Symbolic Analysis & Transformation System (SATS) aims to reduce the complexity of finding test cases to a simple problem of solving a set of equations. Specifically, SATS transforms each path to a set of equations with respect to the input variables; to cover a path, the set of equations must be solved by assigning values to the input variables. This set essentially describes the conditions that lead to the execution of a particular path. SATS works as follows:

Path Localization. Initially SATS utilizes DTDGF's control flow graph, on which a depth first search algorithm extracts all possible paths of the program under testing. Suppose that the control flow graph of a program under testing is *G*, then the set of extracted paths is expressed as $P = \{p_1, p_2, \ldots, p_n\}$, where n is the total number of paths and path p_i, where $1 \le i \le n$, is also a sub-graph of the original graph *G*; note that a path must consist of at least one edge and two vertices. SATS saves each path as a sequence of numbers, e.g. $p_i = \{3,4,5,6\}$, where each number represents a particular node from the nodes of the original control flow graph *G*. Then the user can decide on which paths the generator should run; the user can choose for testing either all paths or a specific path. The next two steps, *Path Selection* and *Symbolization*, are executed consecutively, once for each selected path.

Path Selection. SATS creates a new control flow graph for path p_i. The new control flow graph is a sub-graph of the original graph. Both graphs are stored by the system, while the graph in use is the one representing the selected path; this graph will be analysed in the next step .

Path Symbolization. Symbolic Execution of path step p_i involves transforming all local variables to their symbolic counterpart. A symbolic value is essentially the equivalence of a variable expressed as a function of only the input parameters of the program. This transformation is executed with an up-down algorithm that starts from the first node of the selected control flow graph and symbolically replaces each node's expressions using equivalent expressions that include the input variables. The objective is to reduce the nodes to a set of nodes that include only constraints expressed in relation to the input variables. When all the transformations have been made, the algorithm focuses on remaining nodes, which are path constraints. These constraints are converted to their *TRUE* equivalents; an example is shown in Table 1, where *A* and *B* are conditions, e.g. *A≡X>10* and *B≡Y<20-X, where* variables X and Y are the input parameters.

Table 1: Predicate transformation.

Original Path Constraint	***Path Constraint Converted to its TRUE Equivalent***
A && B ⇒	*!(A&&B) ≡ !A || !B*
A || B ⇒	*!(A||B) ≡ !A && !B*

Test Data Generation. The test data generator utilises a genetic algorithm for each selected path. Note, that if during the execution of a path p_i, a test case is found for another path from the selected list of paths, say p_z, then the test data generator removes path p_z from the list and continues with the rest of the unexecuted paths.

The encoding of a chromosome in the GA represents a solution as a series of *k* genes, with *k* being the number of input parameters embedded in the set of path constraints. The objective is to determine the values of the parameters that can solve the equations. For a given path $p_z \in P$, the fitness function is expressed as

$$f(C, p_z) = \sum_{i=1}^{|PC|} \frac{1}{|PC| + pf(pc_i)} \tag{1}$$

where *C* is the chromosome to be evaluated and *PC* is the set of equations of path p_z, i.e. the path constraints as provided by the symbolic execution; pc_i represents the i^{th} constraint (or equation) and $pc_i \in PC$. The expression $pf(pc_i)$ evaluates each constraint according to the value (i.e. test case) of the chromosome as follows:

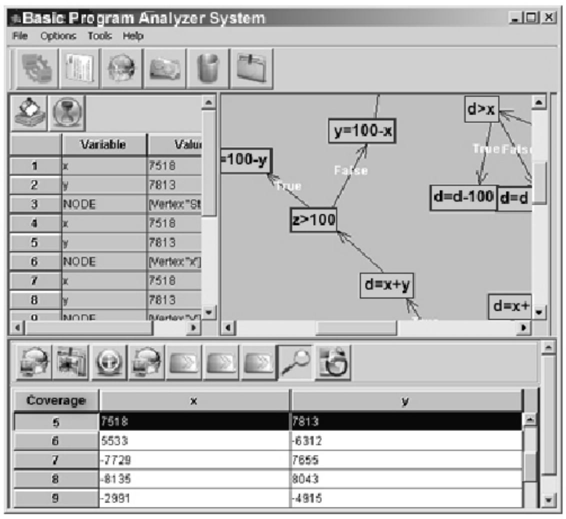
$$pf(pc_i) = \begin{cases} \min(pf(A), pf(A)),\ IFF\ pc_i = (A) \&\& (B) \\ pf(A) + pf(B)),\ IFF\ pc_i = (A) \| (B) \\ ds(A),\ IFF\ pc_i = (A) \end{cases} \tag{2}$$

If path constraint pc_i consists of two predicates connected with the logical operand "AND" (&&) then the value of $pf(pc_i)$ is the minimum between *pf(A)* and *pf(B)*, as pc_i can be evaluated to *TRUE* if and only if each part is evaluated to true. If path constraint pc_i consists of two predicates connected with the logical operand "OR" (||) then the value of $pf(pc_i)$ is the summation of *pf(A)* and *pf(B)*, as pc_i can be evaluated to *TRUE* if and only if at least one of the parts is evaluated to *TRUE*. If path constraint pc_i consists of only one predicate, i.e. $pc_i = A$, then $pf(pc_i)$ is the value of distance *ds(A)*. For *ds(A)*, if the evaluation of A is *FALSE*, then we transform constraint A to the form of C≥0, or C>0, or C≠0, or C=0, e.g., if *A=x>y* then A≡x-y>0. Therefore, *ds(A)* may be expressed as:

$$ds(A) = \begin{cases} 0,\ IFF\ A\ is\ TRUE \\ abs(C),\ IFF\ A \equiv C \geq 0\ or\ A \equiv C = 0 \\ abs(C) + e,\ IFF\ A \equiv C > 0\ or\ A \equiv C \neq 0 \end{cases} \tag{3}$$

SATS starts with the Initial Population Generation, where it generates chromosomes according to the structure of the selected path. Then a repetitive cycle of Evaluation, Reproduction (Crossover & Mutation) and Selection follows. Repetition of computation cycles is terminated either when the maximum number of generations has been reached or a test case that executes completely the path has been found. If the GA fails to find a suitable test case then the path is marked as infeasible (possible dead path). SATS continues to the next path, if there is one, otherwise it terminates the testing process and calculates the coverage.

Figure 2 presents the prototype software application, which was developed to support the whole process. At the right part of this figure the main screen of the application that creates the control flow graph of the program under testing is depicted; the lower part of Figure 2 presents the test cases that were generated. Users can interact with the application and select one or more test cases to inspect both graphically (i.e. on the control flow graph) and numerically (in percentage terms) the coverage achieved.



	Variable	Valu
1	x	7518
2	y	7813
3	NODE	[Vertex:"St
4	x	7518
5	y	7813
6	NODE	[Vertex:"x"]
7	x	7518
8	y	7813

Coverage	x	y
5	7518	7813
6	5533	-6312
7	-7729	7655
8	-8135	8043
9	-2991	-4915

Figure 2: The prototype software application.

Table 2: Experiments on a pool of randomly generated programs with varying LOC and complexity.

AA	LOC	# If statements	Complexity	Our Algorithm		
				edge coverage	condition coverage	edge/condition coverage
1	5	1 multiple 2 single	Low	100%	100%	100%
2	12	1 multiple 2 single	Low	100%	100%	100%
3	14	2 multiple 4 single	Medium	100%	100%	100%
4	11	2 multiple 6 single	Medium	100%	84.61%	92.3%
5	16	3 multiple 12 single	High	100%	87.5%	93.75%
6	16	3 multiple 12 single	High	100%	87.5%	93.75%
7	16	3 multiple 15 single	Very High	100%	73.33%	86.66%

4 EXPERIMENTAL RESULTS

This section presents a list of experiments carried out with the proposed testing framework on a pool of standards and randomly generated JAVA programs. The JAVA programs used in experiments presented in this work can be retrieved from http://www.cs.ucy.ac.cy/~asofok/testing/9.html.

A series of initial trial experiments led us to the following settings: The GA's population size was set to 200 chromosomes, the probabilities of crossover, mutation and switch-mutation's step equal to 0.45, 0.10 and 0.50, respectively, and the maximum number of generations to 1000. 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. The testing framework run on a CENTRINO duo 1.83 GHz with 1.50 GB Ram and JDK 1.5 operating with the Windows XP OS.

As previously mentioned, through the proposed framework the user is able to select one or more paths to produce test data for. The framework runs consecutively a genetic algorithm for each selected path; if a test case that executes the path is found then the framework stores this test case and continues to the next path, if there is one. Testing adequacy was assessed using the *edge*, *condition* and combined *edge/condition* coverage criteria. Edge coverage is calculated as the number of edges executed over the total edges of the control flow graph of the program under testing. To find the executed edges, the framework iterates each test case found to execute a path and adds its executed edges. Likewise, condition coverage is calculated as the average number of conditions evaluated to *TRUE* and conditions executed to *FALSE* over the total number of conditions of the control flow graph of the program under testing. Note that a condition must be executed in order to evaluate to one of the two values (i.e. short circuiting, a state where the first condition of a multiple condition can determine the whole result and therefore the remaining conditions are not evaluated by the virtual machine). Finally, the edge/condition coverage is calculated as the average value of the edge coverage and the condition coverage.

Table 2 shows the first set of experiments that involves seven randomly generated programs varying in terms of complexity expressed in relation to LOC (lines of code), number of conditions, and type and usage (complexity) of conditions (e.g. simple and multiple). The results show that the framework can achieve full edge coverage for every program listed in the table, while the lower condition coverage is 73.33% for the 7th program which is the largest in terms of LOC and the most complicated; complexity is expressed as a function of LOC and conditions.

The second set of experiments compares the performance of the framework against a symbolic testing approach called JCUTE (Sen, Marinov, & Agha, 2005). This set of experiments selected the first three randomly generated programs of Table 2 and applied both approaches. The results of Table 3 show that both frameworks have equivalent performance for the first two programs; as complexity rises, though, our framework clearly outperforms the JCUTE approach as seen in the case

Table 3: Comparison of our algorithm against JCUTE (Sen et al., 2005). Experiments on three randomly generated programs with varying complexity.

AA	LOC	# If statements	Complexity	JCUTE[REF]			Our Algorithm		
				edge coverage	condition coverage	edge/condition coverage	edge coverage	condition coverage	edge/condition coverage
1	5	1 multiple 2 single	Low	100%	100%	100%	100%	100%	100%
2	12	1 multiple 2 single	Low	100%	100%	100%	100%	100%	100%
3	14	2 multiple 4 single	Medium	81.25%	75%	78.125%	100%	100%	100%

Table 4: Comparison of our algorithm against JCUTE (Sen et al., 2005). Experiments on the *TriangleClassification* and the *FindMaximum* standard programs (benchmarks).

Benchmark	LOC	#If statements	Complexity	JCUTE (Sen et al., 2005)			Our Algorithm		
				edge coverage	condition coverage	edge/condition coverage	edge coverage	condition coverage	edge/condition coverage
Triangle Classification	29	9	High	15.78%	14.7%	15.24%	95.73%	88.23%	91.48%
FindMaximum	12	2	Low	85.71%	75%	80.35%	100%	100%	100%

of the third program. In this case, our framework again, achieves full coverage in terms of the edge, condition and combined edge/condition criteria, whereas the other approach fails to reach this level of coverage in any of the aforementioned criteria.

Table 4 shows the third set of experiments, which compares the two approaches over two well known standard programs, the *TriangleClassification* and the *FindMaximum* programs. The results show the efficiency of our framework which manages to outperform dramatically the performance of the other approach. For example, in the case of the *TriangleClassification*, our framework achieves 91.48% edge/condition coverage, whereas the edge/condition coverage of the other approach is only 15.25%. In the case of the *FindMaximum* program, our framework manages to achieve full coverage in each of the three coverage criteria.

5 CONCLUSIONS

This paper presented a framework that combines the advantages of symbolic execution and evolutionary algorithms for solving the complicated problem of generating automatically an adequate set of test cases for a given program written in JAVA. The framework comprises a program analyser, a symbolic executer and a test data generator. The program analyser uses the source code of the program under testing for extracting information, such as variables name and scope, and creating program models, such as control flow graphs. The symbolic executer is responsible for simplifying and transforming a path to a set of path constraints. The test data generator solves the path constraints using genetic algorithms. The user tunes the genetic algorithm by specifying several preferences, such as input parameter boundaries, population size, number of evolutions etc.

Coverage adequacy was assessed using three known criteria, namely *condition*, *edge* and combined *condition/edge* coverage. The proposed testing framework was evaluated using both standard and randomly generated JAVA programs. The results obtained using different sets of experiments demonstrated that the proposed framework performs efficiently on different types of programs in terms of size and complexity. Further results, which compared the performance of our framework against a similar approach revealed the superiority and efficiency of the proposed approach.

Future work will carry out more experiments and will perform more comparisons of our framework. The experiments will use more standard programs richer in LOC and number of conditions. Future work will also consider improving the performance of the symbolic algorithm so as to be able to simplify further the path constraints and allow the test data generator to achieve full coverage even for programs with higher complexity that those listed in Table 2. Currently, we are developing an object oriented model that will be able to capture the features of both object-oriented and graphically depict them on control flow graphs with processing via UML diagrammatical notations. Our future research steps will attempt to incorporate this model into the existing framework and extend the symbolic executer so as to be able to work with the features of the new model.

REFERENCES

Baresel, A., Binkley, D., Harman, M., & Korel, B. (2004). Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis,* Boston, Massachusetts, USA. 108-118.

Baresel, A., & Sthamer, H. (2003). Evolutionary testing of flag conditions. *Lecture Notes in Computer Science 2724: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003),* Chicago, IL, USA. *, 2724* 2442-2454.

Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007): Future of Software Engineering (FOSE '07),* Minneapolis, MN, USA. *, 0* 85-103.

Bottaci, L. (2002). Instrumenting programs with flag variables for test data search by genetic algorithms. *Proceedings of the Genetic and Evolutionary Computation Conference,* New York, USA. 1337-1342.

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

Godefroid, P. (2007). Compositional dynamic test generation. *Proceedings of the 34th ACM Symposium on Principles of Programming Languages (ACM SIGPLAN-SIGACT),* Nice, France. 47-54.

Godefroid, P., Klarlund, N., & Sen, K. (2005). DART: Directed automated random testing. *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05),* Chicago, IL, USA. 213-223.

Kebbal, D. (2006). Automatic flow analysis using symbolic execution and path enumeration. *Proceedings of the 2006 International Conference Workshops on Parallel Processing,* Columbus, Ohio, USA. 397-404.

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM, 19*(7), 385-394.

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

McMinn, P., Harman, M., Binkley, D., & Tonella, P. (2006). The species per path approach to SearchBased test data generation. *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA 2006),* London, UK. 13-24.

Pargas, R. P., Harrold, M. J., & Peck, R. R. (1999). Test-data generation using genetic algorithms. *Journal of Software Testing, Verification and Reliability, 9*(4), 263-282.

Sen, K., Marinov, D., & Agha, G. (2005). CUTE: A concolic unit testing engine for C. *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering,* Lisbon, Portugal. 263-272.

Soffa, M. L., Mathur, A. P., & Gupta, N. (2000). Generating test data for branch coverage. *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00),* Grenoble, France. 219.

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

Sofokleous, A., & Andreou, A. (2008). Dynamic search-based test data generation focused on data flow paths. *Proceedings of the 10th International Conference on Enterprise Information Systems (ICEIS 2006),* Barcelona, Spain. 27-35.

Sofokleous, A. A., & Andreou, A. S. (2008). Automatic, evolutionary test data generation for dynamic software testing. *The Journal of Systems & Software, 81*(11), 1883-1898.

Tillmann, N., & Schulte, W. (2006). Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software, 23*(4), 38-47.

Zhang, J., Xu, C., & Wang, X. (2004). Path-oriented test data generation using symbolic execution and constraint solving techniques. *Proceedings of the Second International Conference on Software Engineering and Formal Methods,* Beijing, China. *, 28* 242-250.