Scientific Journal of Impact Factor (SJIF): 5.71

e-ISSN (O): 2348-4470
p-ISSN (P): 2348-6406

International Journal of Advance Engineering and Research Development

Volume 5, Issue 08, August -2018

A SURVEY OF AUTOMATED TEST DATA GENERATION TECHNIQUES

Sachin D. Shelke[1], Dr. S. T. Patil[2]

[1]*Research Scholar, Department of Computer Engineering,Vishwakarma Institute of Technology, Pune*
[2]*Department of Computer Engineering, Vishwakarma Institute of Technology, Pune*

Abstract-*Software Testing is most labour intensive task and accounts for fifty percent of total cost of development. To reduce high cost of manual testing researchers have tried to automate the software testing process. Test data generation for a given program is the most important problem to be solved in the automated testing process. Many researchers are actively showing the interest in the field of automated test data generation. Many new advancements have been seen in the field of test data generation over past few years. The focus of this paper is to elaborate and review the test data generation techniques used for structural testing. Symbolic execution has received much attention for structural test data generation. Though symbolic execution has many other applications our focus in this paper is to explore the use of symbolic execution in test data generation. Symbolic execution has some limitations when it comes to implement the techniques for real world application like path explore, complex constrains. We discussed the availble solutions to the common problems with the symbolic execution.*

Keywords-*Test data Generation, Structural Testing, Automated Testing, Search Based Techniques, Symbolic Execution*

I. INTRODUCTION

Software testing is very labour intensive process. One way to reduce the testing effors and cost is to automate the process. The automated testing process has three major components to look at,Test data generation, test case execution and result matching. Test data generation is process of identifying the input which satisfy the testing criteria. These inputs are then passed to the program in order to evaluate the internal structure of the program. This is commonly known as structural or white box testing. Structural coverage criteria require the certain elements of system to be evaluated like specific branches, loops, conditions etc. The other commonly used technique is functional test where focus is to check the external characteristics of the system like correctness, stress capabilities, security etc without looking into the internal structure of the program. Figure 1. shows the overview of the broad categories and different approaches used for Automated Test Data Generation(ATDG)[2].

1.1. Random Testing

It is a simple technique for random selection of test cases. Test cases are randomly chosen based on operational domain. Random Testing simply executes the program with random inputs and then observes the program structures executed. This technique works well for simple programs. However structures that are only executed with a low probability are often not covered. There are many methods to distribute the random input across the complete domain which helps to improve the effectiveness of the random test data generator.
T.Y. Chen, H. Leung and I.K. Mak[1] introduced an enhanced form of random testing called Adaptive Random Testing. Adaptive random testing seeks to distribute test cases more evenly within input space. In order to distribute test cases evenly across the domain newly generated test case should not be too close to any of the previously generated ones. This is achieved through use of two sets of test cases, namely the executed set and the candidate set which are disjoint. Initially executed set is empty and we go on adding the test cases which are exucuted without revealing the defects in the program. While candidate set is the set of randomly selected test cases. Then we update the executed set gradually by selecting test cases from the other set i.e. candidate set. From the candidate set, an element that is having large distance from the executed test cases, is selected as the next test case. Andrea Arcuri, Per Kristian Lehre and Xin Yao[3] review and analyze the debate about random testing. Its benefits and drawbacks are discussed. Authors proved the use of random testing in real world environment and also suggested the further development areas in the field of random testing. Ali Shahbazi, Andrew F. Tappenden and James Miller proposed Random Border CVT (RBCVT), which helps to improve the existing random test techniques. This is achieved through the generation of random inputs depends upon certain

mathematical calculations, which helps to cover the entire work space. Many tools like RANDOOP[4] are available which use the random test case generation techniques of some form.
We can summarize from the above information that the random testing is simple and till effective form of test data generation techniques which can be even applied to the real world project, though certain improvements and chanllenges need to be handled.

1.2. Path Oriented Testing

In path testing we first find the all possible paths for the given program. Paths can be easily decided by using Control Flow Graph(CFG). Basic information regarding the CFG in given in the next section. Here challenge is to find the infeasible path, paths those cannot be covered at all by any input. Many algorithms have been proposed in order to decide the infeasible paths for a given program. Symbolic execution[22] is the effective and moste commenly used technique for finding the infeasible paths. In this techniques we check the internal structure of the program i.e. white box testing, where small units of the program are checked for specific errors like typographical, syntactical etc. This technique is known as unit test.
XING Ying, GONG Yun-zhan, WANG Ya-wen, ZHANG Xu-zhou[5], Liu Shimin, Wang Zhangang[6] have contributed in the field of path base test data generation.

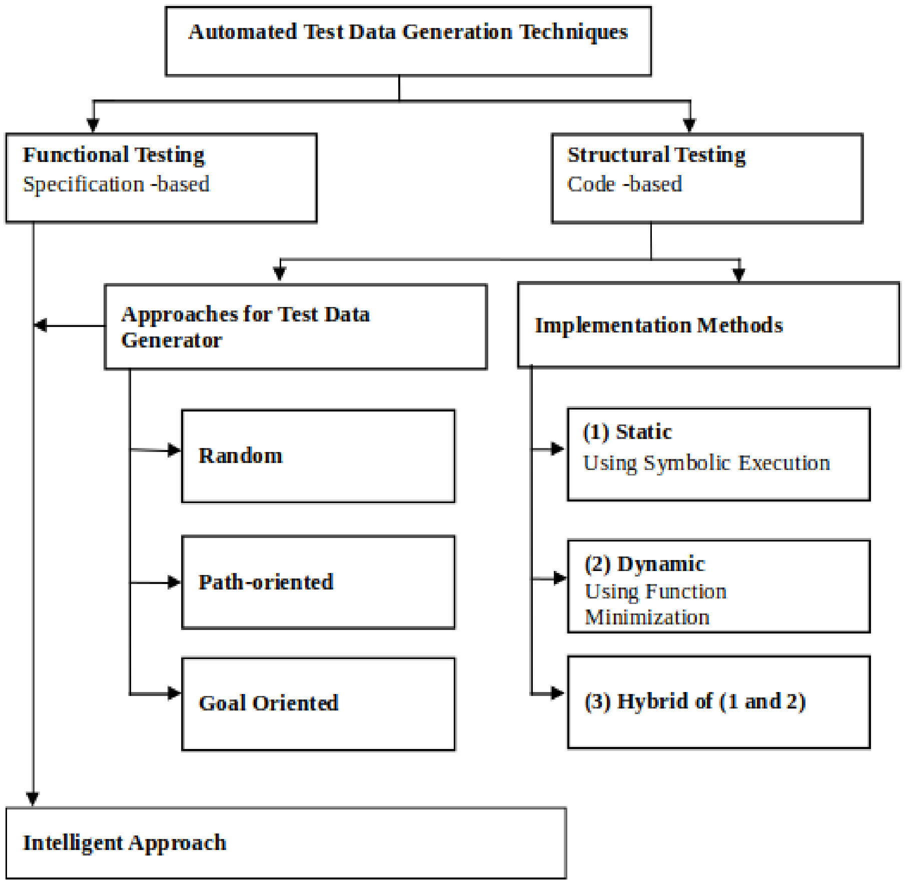

Figure 1. An overview of ATDG techniques[2]

1.3. Goal Oriented Testing

All of the test data generation techniques concentrate on the execution of a path. Path testing not only includes execution of each statement but it should also consider the different paths, decisions and branch coverage. Goal-Oriented approach takes some different path. It removes the requirement, that each path should get executed, rather it focus on a specific goal and in order to achieve goal it may take select any path/branch. There are many techniques proprosed[[8] in order to

select the path that will help to aheive the goal. One of them to divide the CFG branches into critical, semi-critical or non-essential and then focus on the critical branches so the focus should not get away from the critical branches.
TheAnh Do, Siau-Cheng Khoo, Alvis Cheuk Ming Fong, Russel Pears, Tho Thanh Quan[7] presented a goal oriented testing approach for effectively and efficiently exploring security vulnerability errors. A goal is a potential safety violation and the testing approach is to automatically generate test inputs to uncover the violation.

II. BASIC CONCEPTS

A test data generation problem can be defined as function P: S → R, Where, P: A module/program under test S: Set of all possible inputs and R: Set of all possible outputs. P(x) denotes execution of program/module P for certain input x. A Control Flow Graph of a program P can be represented as a directed graph as, G= (N, E, s, e) Where, G: Directed Graph, N: Number of nodes in a directed graph, E: Number of edges denoted as, E={(n,m)| n,m ϵ N}, s: Start node, e: End node Each node denotes block which is set of instructions. Block is executed as a whole. Edge in a graph shows the execution flow from one node to other. Condition node has more than one outgoing edges like node 1 in Figure 2. such edges are called as branches. Figure 2. shows the control flow graph for the commonly used trianle program(Table 1.) in testing scenario. The triangle classification program is a benchmark used in many testing papers. Assuming three non-zero, non-negative integer lengths for the sides of atriangle, the program decides if the triangle is isosceles, scalene, equilateral, or

Table 1. Triangle Program:

```
int tri_type(int a, int b, int c)
            {
                  int type;
1                  if (a > b)
2-4               { int t = a; a = b; b = t; }
5                 if (a > c)
6-8               { int t = a; a = c; c = t; }
9                 if (b > c)
10-12                    { int t = b; b = c; c = t; }
13                 if (a + b <= c)
                  {
14                       type = NOT_A_TRIANGLE;
                  }
                  else
                  {
15                type = SCALENE;
16                      if (a == b && b == c)
                        {
17                       type = EQUILATERAL;
                        }
18                      else if (a == b || b == c)
                        {
19                       type = ISOSCELES;
                        }
                  }
e                 return type;
            }
```

invalid. Nodes corresponding to decision statements (for an example an if or a while statement) are referred to as branching nodes. In thetriangle example, branching nodes are nodes 1, 5, 9, 13, 16 and 18. Outgoing edges from these nodes are referred to as branches[8].

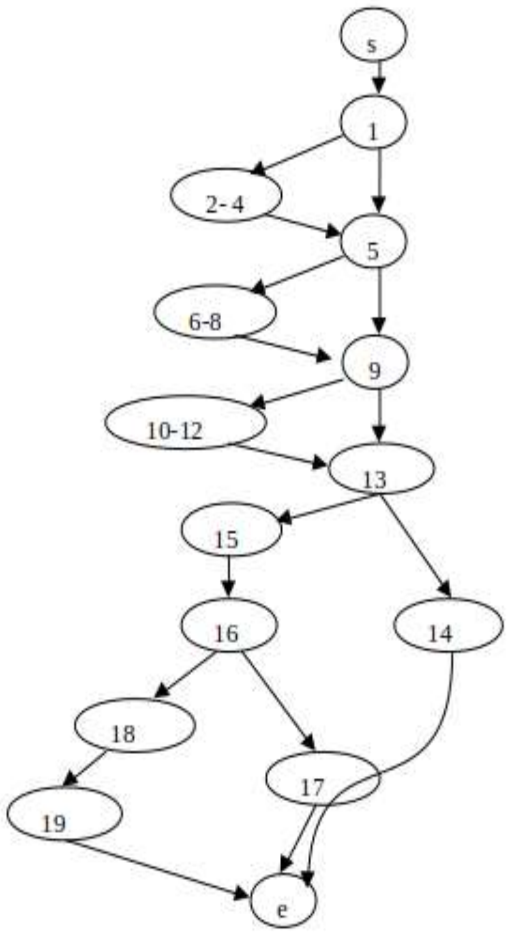

Figure 2. Control flow graph for triangle program

A control flow for the trianlge program is shown below in Figure 2. where, s: Start node, e: End node and remaining node represents the statement/ block of statements those get executed as result of branch decision. A path P through a control flow graph is a sequence P =< n 1 , n 2 , . . . , n m >,such that for all i, 1 ≤ i < m, (n i , n i+1) ∈ E. A path is said to be feasible if there exists a program input for which the path is traversed, otherwise the path is said to be infeasible.

III. TEST DATA GENERATOR

Test data generator uses program analyzer for analysing the program so that we can create the, dependence graph which will be used by path selector to select a particular path dependence upon the strategy used for path selection. Constraints will be generated for the selected path and test data will be generated to solve these constraints. Figure 3. shows the architecture of test data generator[9]. Detailed description of this process is given below in this section.

3.1.1. Program Analyzer

Program analyzer takes the program as a input and produces the information that can be utilized by path selector and test data generator. Program analyzer uses the one or more than one commonly used techniques like CFG, data dependence graph, finite state machines etc. Architecture of test data generator is shown in Figure3. It basically consist of program analyzer, path selector, contraint generator and constraint solver. Program analysis helps to select the specific path for which constraints are generated by the contraint generator. We need to find the data that will solve the generated constraints. This data is considered as input for the program in order to reveal the defects. The architecture shown in Figure 3. is well suited for the programs having loop and other constructs like arrays. The output from the program analyzer is given to the path selector.

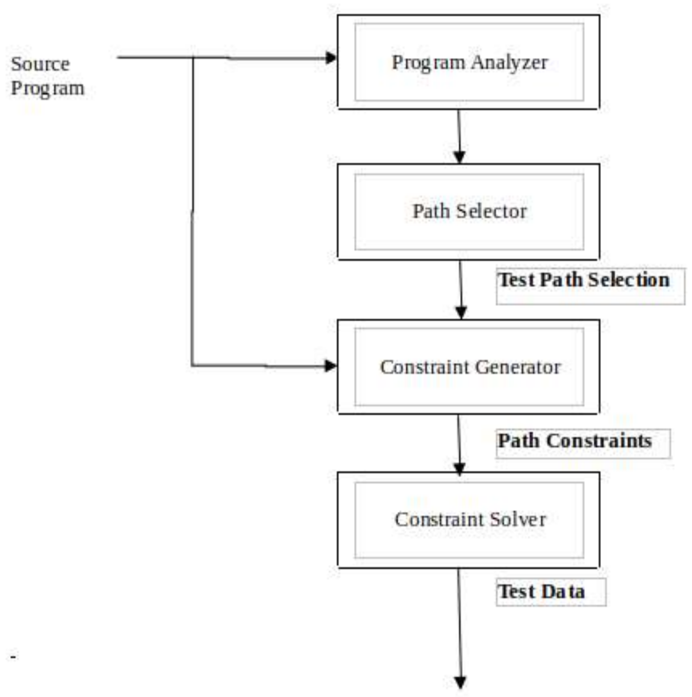

Figure 3. Architecture of test data generator

3.1.2. Path Selector

The path selector helps to indentify the path inorder to achieve the testing goal. The path selector selects the set of paths those basically convers the entire structure of the program removing the need of executing all the paths of given program as the number of possible path may be very large and even some of them may be infeasible. Outpute generated from the path selector i.e. set of basis path is passed to the constraint generator which generates the constraints for the selected paths. This pahse is not considered in many recent test data generator tools.

3.1.3. Test Data Generator

The test data generator use the information generated by the program analyzer and path selector in order to generate the test data. Once path is selected test data generators task is to generate the data that will traverse the selected path. If generated data takes away the control from the selected path, then values/data need to be updated. This can be done using many methods like search methods, simulated annealing, hill climbing and different heuristic techniques. These mehods update the vaules in order to keep the control close to the selected path and traverse the same.

IV. IMPLEMENTATION METHODS

Static method of test data generation is based on analysis of internal structure of the program without executing the program. Whereas dynamic methods tend to run the program with the test data in order to evaluate the internal structure.

4.1. Symbolic Execution

Symbolic Execution is not the execution of a program in its true sense, but rather the process of assigning expressions to program variables as a path is followed through the code structure. The technique can be used to derive a constraint system in terms of the input variables which describes the conditions necessary for the traversal of a given path[8].Although the symbolic execution technique was proposed early in the seventies, the technique has got much attention in recent years due to two reasons. First, the application of symbolic execution on large, real world programs requires solving complex and large constraints. [10]. During the last decade, many powerful constraint solvers [11], Yices [12], STP[13] have been developed. These contraint solvers helped to wide spread the use of symbolic applicatons in many different areas[10]. Second, It takes high computation time in order to solve the complex contraints generated by the constraint generator and it was not possible to solve those complex constraints with the computing capacity available at that time. Moreover someof the constraints could not be solved due lack of information. However, today's commodity computers are arguably more powerful than the supercomputers of the eighties[10].

Shay Artzi, Adam Kie zun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst[14] developed the tool called Apollo which generates test inputs for a Web application, monitors the application for crashes, and validates that the output conforms to the HTML specification. The technique used in this tool generates tests automatically, runs the tests capturing logical constraints on inputs, and minimizes the conditions on the inputs to failing tests so that the resulting bug reports are small and useful in finding and fixing the underlying faults. Nicolas Rosner, Jaco Geldenhuys, Nazareno M. Aguirre, Willem Visser and Marcelo F. Frias[15] presented bounded lazy initialization with SAT support (BLISS), a novel technique that refines the search for valid structures during the symbolic execution process. This technique basically helps to prune the redundant partially symbolic structures that could not be pruned by the previously introduced techniques. Anh Do, Siau-Cheng Khoo, Alvis Cheuk Ming Fong, Russel Pears, Tho Thanh Quan[7] presented a goal-oriented testing approach for effectively and efficiently exploring security vulnerability errors. A goal is a potential safety violation and the testing approach is to automatically generate test inputs to uncover the violation. Authors here presented A path exploration algorithm that exploits data dependencies to perform dynamic symbolic execution for effectively and efficiently uncovering desired test goals. A. Jeerson Outt, Zhenyi Jin and Jie Pan ISSE Technical report. Further advancement in the symbolic execution uses the type-dependence analysis to minimize the instrumentation efforts[17].

Cadar et al.,[18] explained the use of symbolic execution in the field of improved code coverage and expose of the software bug from the system under the test. Castro et al.,[19]explained the use of symbolic execution in the field of privacy preserving and error reporting. Zhang et al.,[20] show the use of symbolic execution in load testing. Load testing is basically carried out to check the system capabilities and limitations. Use of symbolic execution for testing of graphical user interface is explained by Ganov et al.,[21]. Many symbolic execution tools are publically available like Symbolic Pathfinder[22], JCUTE[23], CUTE[24], Klee[18], S2E[25], and Crest. Several other approaches implement dedicated tools to perform various based on some form of symbolic execution. [26,33,35].

Symbolic execution traditionally used for checking sequential programs. Many limitations were encountered even with the sequential program like forward reference values, use of pointers. Further difficultied include the handling of procedure calls.

4.2. Problems with Symbolic Execution

4.2.1. Memory. problems like pointer handling, array references and other complex objects need to be handled by using symbolic execution.

4.2.2. State Space Explosion. while testing real world software, we may have large number of paths and even these paths may grow exponentially if we consider the language constructs like loop, decisions etc. Symbolic executin of these paths may require high computational overhead. So it is unlikely that technique of symbolic execution will explore all the stated paths for the given program/software.

4.2.3. Legacy Softwares. For old software systmes we may not have the complete source code available, which then it difficult to generate the executable paths, which is the requirement for constraint solver.

4.2.4. Complex Constraints. It may not always be possible to solve path constraints. it is possible that the computed path constraints become too complex to using available constraint solvers Conclusion Symbolic execution techniques have evolved significantly in the last decade, with notable applications to compelling problems from several domains like software testing.The fundamental problems that the technique suffers from are a long-standing open problem More research is needed to improve the technique's usefulness on real-world programs. However, it can be used in combination with those other techniques like dynamic execution, search-based testing.

Solutions identified to some extentd to the above mentioned problmes are listed below. These solutins help to limit the above mentioned problems to some extend. First solution is mixing of symbolic and concrete use of symbolic execution. In classical symbolic execution it is not possible to find the constraints that cannot be solved by using the constraint solver. This can be handled through the mixing of above mentioned techniques. Second solution to the above mentioned problems is to use the dynamic symbolic execution (DSE) or dynamic test generation [7,29], is to have concrete execution drive symbolic execution. This technique helps to remove many problems associated with the symbolic execution. Initially we randomly select the input values. We use the classical symbolic execution as long as we execute the path which we are intended to do. As we are moving away from the inteded path we update the values using concrete execution. Here we can can choose one out of many availbale techniques like alternating variables, heuristics search methods, search methods, finite state machines or combination of one or more of stated techniuqes. Another similar approach is combination of symbolic and concrete execution of some form.

V. CONCLUSION

This paper presents the survey of promising techniques used for automated test data generation, including symbolic execution, random and adaptive random tesing techniques. We also classified these techniques, based upon the general approches of software testing, into functional and structural category. This survey shows that the random testing is good option in some conditions and this technique should be seriously considered for use in some specific applications. There is much research in the field of symbolic execution over the last decade, with widespread applications to compelling problems from several domains like software testing (e.g., improve code coverage and expose software bugs), security (e.g., generation of security exploits), code analysis and testing of graphical user interface. This trend has not only improved existing solutions, but also led to novel ideas.
We also have discussed key aspects and challenges to the symbolic execution and some available and possible solutions to the problems associated with the symbolic execution. We hope this will help and inspire the researcher in the futher research in the field of automated test data generation.

REFERENCES

[1] T.Y. Chen, H. Leung and I.K. Mak, “Adaptive Random Testing”, ASIAN 2004, LNCS 3321, pp. 320–329, 2004.
[2] Shahid Mahmood, “A Systematic Review of Automated Test Data Generation Techinques”, Master Thesis, Software Engineering, Thesis no: MSE-2007:26 October 2007.
[3] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand, “Random Testing Theoretical Results and Practical Implications”, IEEE Computer Society, Dec 2011.
[4] Carlos Pacheco and Michael D. Ernst, “Randoop Feedback Directed Random Testing for Java”, OOPSLA 2007 Montreal, Canada, October 21-25, 2007.
[5] XING Ying, GONG Yun-zhan, WANG Ya-wen, ZHANG Xu-zhou, “Intelligent Test Case Generation Based on Branch and Bound”, The Journal of China Universities of Posts and Telecommunications, April, 2014.
[6] Liu Shimin, Wang Zhangang, “Genetic Algorithm and Its Application in the Path-Oriented Test Data Automatic Generation”, Elsevier Ltd, 2011.
[7] Anh Do, Siau-Cheng Khoo, Alvis Cheuk Ming Fong, Russel Pears, Tho Thanh Quan, “Goal-Oriented Dynamic Test Generation”, Journal of Information and Software Engineering, June 2015.
[8] Phil McMinn, “Search Based Software Test Data Generation: A Survey”, Software Testing, Verification and Reliability, 14(2), pp. 105-156, June 2004.
[9] Hitesh tahbildar and bichitra kalita, “Automated Software Test Data Generation Direction of Research”, International journal of computer science & engineering survey (ijcses) vol.2, no.1, feb 2011.
[10] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, “An Orchestrated Survey of Methodologies for Automated Software Test Case Generation”, The Journal of System and Software”, April 2013.
[11] de Moura, L.M., Bjorner, N., “Z3: An Efficient SMT Solver”, Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’08), pp. 337–340, 2008.
[12] Dutertre, B., de Moura, L., “A Fast Linear-arithmetic Solver for DPLL(T)”, In Proceedings of the 18th International Conference on Computer Aided Verification (CAV’06), pp. 81–94, 2006.
[13] Ganesh, V., Dill, D.L., “A Decision Procedure for Bit-Vectors and Arrays”, Proceedings of the 19th International Conference on Computer Aided Verification (CAV’07), pp. 519–531, 2007.
[14] Shay Artzi, Adam Kie zun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst, “Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking”, IEEE Transactions On Software Engineering, VOL. 36, NO. 4, JULY/AUGUST 2010.

[15] Nicolas Rosner, Jaco Geldenhuys, Nazareno M. Aguirre, Willem Visser and Marcelo F. Frias, “BLISS: Improved Symbolic Execution byBounded Lazy Initialization with SAT Support”, IEEE Transactions On Software Engineering, VOL. 41, NO. 7, JULY 2015.

[16] Anh Do, Siau-Cheng Khoo, Alvis Cheuk Ming Fong, Russel Pears, Tho Thanh Quan, “Goal-Oriented Dynamic Test Generation”, Journal of Information and Software Technology, June 2015.

[17] Kuo-Chung Tai and Yu Lei, “A Test Generation Strategy for Pairwise Testing”, IEEE Transactions On Software Engineering, VOL. 28, NO. 1, JANUARY 2002.

[18] Cadar, C., Dunbar, D., Engler, D.R., “KLEE: Unassisted And Automatic Generation of High-Coverage Tests for Complex Systems Programs”, Proceedings of the Symposium on Operating Systems Design and Implementation, pp. 209–224, 2008.

[19] Castro, M., Costa, M., Martin, J.P., “Better Bug Reporting With Better Privacy”, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’08), pp. 319–328, 2008.

[20] Zhang, P., Elbaum, S.G., Dwyer, M.B., “Automatic Generation of Load Tests”, Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE’11), pp. 43–52, 2011.

[21] Ganov, S.R., Killmar, C., Khurshid, S., Perry, D.E., “Test generation for graphical user interfaces based on symbolic execution”, Proceedings of the 3rd IEEE/ACM International Workshop on Automation of Software Test (AST’08), pp. 33–40, 2008.

[22] Pasareanu, C.S., Rungta, N., “Symbolic Pathfinder: Symbolic Execution of Java Bytecode”, Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE’10), pp. 179–180, 2010.

[23] Sen, K., Agha, G., 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools”, Proceedings of the 18th International Conference on Computer Aided Verification (CAV’06), pp. 419–423, 2006.

[24] Sen, K., Marinov, D., Agha, G., “CUTE: A Concolic Unit Testing Engine for C”, Proceedings of the 2005 Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 263–272, 2005.

[25] Chipounov, V., Kuznetsov, V., Candea, G., “S2e: A Platform For In-Vivo Multi-Path Analysis Of Software Systems”, Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS’11), pp. 265–278, 2011.

[26] Corina S.Pasareanu and Willem Visser, “A Survey Of New Trends in Symbolic Execution for Software Testing and Analysis”, Int J Softw Tools Technol Transfer (2009) 11:339–353, 2009.

[27] Burnstein, "Practical Software Testing", Springer, ISBN 0-387-95131-8.

[28] Juha Itkonen, Mika V. Mantyla, Casper Lassenius, "The Role of the Tester’s Knowledge in Exploratory Software Testing", IEEE Transactions on Software Engineering, VOL. 39, NO. 5, MAY 2013.

[29] Sreedevi Sampath, Renee Bryce, Atif M. Memon, "A Uniform Representation of Hybrid Criteria for Regression Testing", IEEE Transaction on Software Engineering, Vol. 39, NO. 10, October 2013.

[30] Phil McMinn, Mark Harman, Kiran Lakhotia, Youssef Hassoun, and Joachim Wegener, "Input Domain Reduction through Irrelevant Variable Removal and Its Effect on Local, Global, and Hybrid Search-Based Structural Test Data Generation", IEEE Transaction on Software Engineering, Vol. 38, NO. 2, March/April 2012.

[31] E.K. Burke and G. Kendall, “Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques”. Springer 2006.

[32] Victor R. Basili, Richar W. Selby, "Comparing Effectiveness of Software Testing Strategies", IEEE Transaction on Software Engineering, Vol. 12 Dec 1987.

[33] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger, “PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis”, CEA, Saclay.

[34] D. Whitley, "An overview of evolutionary algorithms: Practical issues and common pitfalls", Elsevier, Information and Software Technology, 43(14):817-831, 2001.

[35] Hwa-You Hsu and Alessandro Orso, "MINTS: A General Framework and Tool for Supporting Test-suite Minimization", IEEE ICSE’09, Vancouver, Canada, May 16 - 24, 2009.