# Symbolic Execution in Software Testing

Xusheng Xiao xxiao2@ncsu.edu
Xi Ge xge@ncsu.edu
Da Young Lee dlee10@ncsu.edu

## Abstract

*Symbolic execution is a way to track programs symbolicly rather than executing them with actual input value. With the impressive progress in constraint solvers, concolic path-based testing tools have literally blossomed up by combining both concrete and symbolic execution, which makes it possible to perform automatic path-based testing on large scale programs. However, these technologies are still suffering from the path explosion problem, since achieving 100% path coverage is to enumerate all paths between two nodes in a graph, which is well known as a NP-hard problem. To alleviate this path explosion problem and to reduce the computation complexity, different techniques has been proposed, such as pruning search space of symbolic execution, selective symbolic execution and fitness-guided path exploration. In this report, we provide the details on the problem of test data generation and present three techniques to alleviate this path explosion problem. We also discuss the early researchers who contributed to the field of research of test data generation using symbolic execution.*

## 1 Introduction

To achieve automatic test (data) generation from source code, there are currently two well established frameworks: constraint-based testing [1, 2, 13, 28, 22] (CBT) and search-based testing [20, 21, 15, 17, 18] (SBT). CBT approach focuses on translating part of a program into a logical formula whose solutions are relevant to test data, while SBT approach is based on exploring the input space of the program using optimisation-like methods to guide the search toward relevant test data. In [15], Harman et al. show that SBT can be local or global. Similarly, CBT also can be global, translating the whole program in a formula [1, 2], or local (path-based) [13, 28, 22], focusing on a single path. In this report, we focus on path-based CBT, referred as path-based testing. Obviously such a local analysis is not sufficient to prove correctness of the whole system, however it is sufficient to derive a test data exercising the given path at runtime. Then, iterating the process on many different paths allows to automatically build test suites achieving a given structural coverage objective, e.g. branch coverage.

Symbolic execution [16] is a way to track programs symbolicly rather than executing them with actual input value. Concolic path-based testing tools have literally blossomed up recently [25, 4, 7, 8, 23, 29] with the impressive progress in constraint solvers. Concolic path-based testing tools combine both concrete and symbolic execution (referred as concolic execution [13, 28] or mixed execution [7]), which makes it possible to perform automatic path-based testing on large scale programs. By executing the program under test with concrete values while performing symbolic execution, symbolic constraints on the inputs can be collected from the predicates in branch statements, forming an expression, called path condition. To explore new paths, part of the constraints in the collected path conditions are negated to obtain new path conditions, which are sent to a constraint solver to compute test inputs for new paths. In theory, all feasible execution paths will be exercised eventually through the iterations of constraint collection and constraint solving in DSE.

A program under test can be modeled as a control flow graph (CFG) [3], whose nodes represent simple primitive statements (such as input, output, and assignment) and edges represent the flow of control. An execution path of the program is a path on CFG from the starting node, entry of the program, to the exit node, exit of the program. Thus, to explore all paths of the program under test, i.e. achieving 100% path coverage, is to enumerate all paths between two nodes in a graph, which is well known as a NP-hard problem [14]. To alleviate this path explosion problem and to reduce the computation complexity, different techniques has been proposed, such as pruning search space of symbolic execution [5], selective symbolic execution [9] and fitness-guided path exploration [30]. In this report, we provide the details on the problem of test data generation

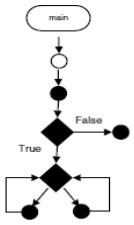


**Figure 1. Example of Look-Ahead (LA) Heuristic**

and present three techniques to alleviate this path explosion problem. We also discuss the early researchers who contributed to the field of research of test data generation using symbolic execution.

## 2 Pruning Search Space of Symbolic Execution

Recent and impressive progress in constraint solvers as well as the combination of both concrete and symbolic execution (referred as concolic execution [28, 12] or mixed execution [7]) make it possible to perform automatic path-based testing on large scale programs. However, these technologies are still suffering from two major bottlenecks: efficient constraint solving and the path explosion phenomenon. Sébastien Bardin and Philippe Herrmann [5] focus on the second issue and propose three complementary heuristics geared toward lowering path explosion. All these heuristics deal with different distinct sources of path explosion.

To cover all paths of a program is not the primarily objective of current testing practices. Often the case, it is only required to fully cover a class of structural artifacts of the program code source, such as statements, branches or atomic predicates. In the rest of the section, we denote these three classes of artifacts as structural coverage. There is an obvious mismatch between path-based approaches and such item coverage goals: while each new test data does cover a new path, it may hit no new item. Thus, path-based testing methods tend to waste a lot of time trying to compute irrelevant test data, i.e. test data exercising no new structural coverage.

To address the path explosion issue in path-based testing with item coverage objectives, they provide three heuristics to discard irrelevant paths as much as possible, reducing of the number of solver calls and the whole computation time. The three heuristics are used as enhancements of a (bounded) depth-first search (DFS) path-based procedure, either purely symbolic or concolic. Original path-based testing techniques were based on DFS [13, 28, 22], while some recent works advocate using other search strategies [8, 19, 23].

### 2.1 Look-Ahead heuristic

The key idea of the Look-Ahead (LA) heuristic is to perform a reachability analysis in terms of reachable items in the CFG, and decide whether the current path must be expanded based on the reachability analysis. If no new items can be reached, then exploration along the current path is stopped.

Figure 1 shows an example for illustrating the Look-Ahead (LA) heuristic. Let us assume the depth bound $k \geq 4$ and the objective is to achieve full statement coverage, and every path of the program is feasible. Then the original DFS based produre needs to explore $\approx 2^k$ paths to achieve full coverage (because of the two nested loops) while DFS+LA requires at most 3 paths: one path to cover the false branch and two paths to cover two loops.

### 2.2 Max-CallDepth (MCD) heuristic

The major source of path explosion is function calls, and especially nested function calls. It is more embarrassing when only the top-level function is of interest. For example, the procedure may explore alternative (long) paths due to backtrack in deep callees while a simple backtrack at top-level would be sufficient. Example 1 of figure 2 gives such a behavior. Figure 3 shows a small program where DFS achieves full branch coverage of main function with $k \geq 8$. DFS+MCD with mcd = 0 and any value of $k$ cannot cover one of the two branches of the main function, since a backtrack in sub-function $f$ is necessary. Consider the program of Figure 2, take mcd = 0 and let us assume that sub-function f does not affect variable b. Then both DFS+LA+UT and DFS+MCD achieve full branch coverage. However, DFS+MCD explores only the two paths of function

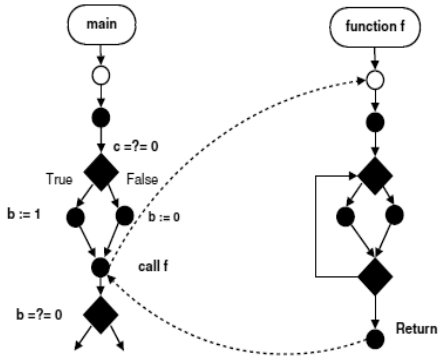


**Figure 2. Example 1 of Max-CallDepth (MCD) heuristic**

main, while DFS+LA explores $\approx 2^k$ paths. DFS+LA+MCD performs better than DFS+LA in the example of Figure 2, and it outperforms DFS+MCD in the example of Figure 1.

The principle of the Max-CallDepth heuristic (MCD) is to prevent backtracking in deep nested calls, hoping such a deep decision is not mandatory to cover the function under test. It is clear that this heuristic makes sense only in unit testing. Moreover, contrary to LA, MCD may discard relevant paths and prevent the full coverage of the function under test.

### 2.3 Solve-First (SF) heuristic

Compared to other graph searches, DFS path exploration lowers memory consumption, since only one path at a time has to be maintained. However, DFS has at least two disadvantages in path-based testing. First, in a realistic environment where the number of tests we can run is limited, DFS may focus only on a very deep and narrow portion of the program under test and achieve only a poor global coverage. Second, DFS-based procedures explore and try to solve first the longest path prefixes, while shorter prefixes with probably simpler constraints may have covered the same items. Hence, DFS suffers from a slow initial coverage-speed and may waste resources on unduly complex paths.

Solve-First heuristic (SF) is to partially address both issues described above, while keeping most of the advantages of DFS. SF is a DFS with the slight following modifications. When the procedure reaches a branch (choice point) for the first time, the first successor node is chosen as usual but all alternative successors are immediately resolved. Test data are derived from the solutions and executed in concrete mode to update coverage information. All solutions found are stored in a cache. On backtrack, once the next successor is chosen, the corresponding test data is retrieved from the cache. The test data is then relaunched and the procedure proceeds as usual in the concolic case.

The program shown in Figure 4 is parametrized by its number of nodes 2n + 1. Let us assume that all paths are feasible and that the first concrete path goes through every true branch. DFS+LA needs to explore n + 1 paths to achieve full instruction coverage while DFS+LA+SF needs only to explore 2 paths. The depth bound for the concrete execution may be much larger than the one for concolic execution, allowing to cover more items. Note that essentially SF is concolic and has no interest in a symbolic setting.

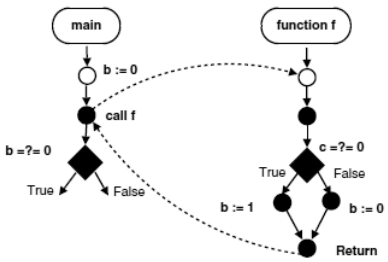


**Figure 3. Example 2 of Max-CallDepth (MCD) heuristic**

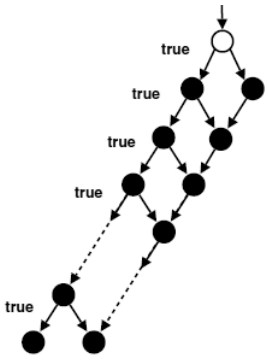


**Figure 4. Example of Solve-First (SF) heuristic**

### 2.4 Conclusion and Related Work

These three heuristics address the path explosion issue in common cases of irrelevancy. All these heuristics are lightweight, both in terms of implementation cost and computational overhead, and they are all complementary since each one addresses a typical source of path explosion. Their evaluations show that these heuristics have a significant impact on the number of paths considered and overall performances. Although these three techniques have been presented in a DFS framework, they are easy to adapt to other path-based frameworks.

Other related works address the problem of path explosion in rather different ways than their heuristic approach. Path explosion due to thread interleaving in concurrent programs is addressed in [27, 11]. The first work is inspired by partial orders methods from (explicit) model checking, while the second work is more ad hoc. Going back to nested function calls, some works have been conducted on modular test generation in order to avoid function inlining. Function summaries are used by recent approaches, which can be either discovered during the static analysis [26, 12] or provided manually [24]. Functions can also be handled lazily [26].

## 3 Fitness

Structural software testing aims at achieving full or high code coverage such as statement and branch coverage of the program under test. The problem of testing for finding bugs could be reduced to the problem of structural testing that achieves full code coverage. A certain defected area of the source code could be considered as a statement guarded by a condition on the input values. For example, the negation of the assertion condition will witness a bug.

Random testing is one of the most commanly used technique for structural software testing since its easy implementation and the marginal overhead in choosing inputs. It simply generates the test input randomly and feed the input to the program under test. Even though the unbiased and effiecient nature of random testing, it will face difficulty to cover a certain statement or branch whose execution requires strict conditions on the input. For example, to cover the statement in line 6 of the example method in figure 1, the integer value of x needs to be exact 90, and the array of y contains more than 110 elements whose value are 15. This harsh condition on the input values makes it almost impossible to reach the statement in line 6.

```
public bool TestLoop(int x, int[] y) {
1   if (x == 90) {
2     for (int i = 0; i < y.Length; i++)
3        if (y[i] == 15)
4           x++;
5     if (x == 110)
6        return true;
    }
7   return false;
}
```

**Figure 5. Figure 1. An example method under test.**

To address this issue faced by the random testing, dynamic symbolic execution(DSE) that systematically explores feasible paths of the program under test has been proposed recently. DSE executes the program for a given input, and performs collection of symbolic constraints on the inputs implied by predicate statements along the execution. The conjunction of all the conditions will be called path condition. DSE increases the code coverage by iteratively flipping the branch node of an already explored path and using constraint solver to generate test inputs based on the new path condition if it is satisfiable. For example, assuming the initial argument of x and y in the example method are 0 and $\{0\}$, respectively, the false branch of the line 1 statement, x!=90, is taken. Negating the branch will give the input value x=90 and y=$\{0\}$ to cover the true branch of line 1.

Even though DSE greatly improves the efficiency of achiving higher code coverage. It still faces multiple challenges in practical use. The covering of the true branch of line 5 needs exactly 20 executions of the true statement of line 3 inside the loop. By using DSE to hunt such a case, we would explore at least $2^{20}$ different execution paths before we can reach line 6 (using random searching strategy). This is in NP-Hard that renders the full coverage of the method almost impossible.

There are several other type of searching strategies besides random searching. DART [13]and CUTE [28] adoped the depth-first searching strategy and EXE [8]provide either depth-first search strategy and the mixture of best-first and depth-first strategy based on the code coverage heuristics. A generational search that explores only a very limited horizon are usd by SAGE [23].The drawbacks of all the searching strategies are biased nature that favor particular control flow points, e.g, the depth-first searching favors the final branches and the width-first searching strategy favors the first branches. None of these ways are capable of reaching target test stated in the example method efficientlly.

To enhance the efficiency of DSE in achieving higher code coverage, the authors has proposed a novel way to guide DSE called fitnex, which is a searching strategy that uses state-dependent fitness values to guide path exploration. Through the fitness-guided path exploration, the full code coverage in cases like the example method under test will be more easily achieved. The procedure to use Fitnex strategy on the example method showed in figure 1 will be presented as follows:

**Fitness computation for a path.** The fitness values are used to decide which branch's flipping would be more helpful in covering the target statement. Extracted from predicate of the conditions in executing test target, fitness functions are used to compute the fitness value, reflecting how close a path's execution is to the covering of the test target. The exploration then honors the fittest path which is the closest to the covering of the test target. The fitness function for the example method is $f(x) = |110 - x|$. The smaller the fitness function value, the closer the path's execution is to cover the target. Considering the following 5 execution path,

| Test | procedure | Path |
|---|---|---|
| 0 | TestLoop(0, new int[] 0); | Path 0: 1F |
| 1 | TestLoop(90, new int[] 0); | Path 1: 1T, 2T, 3F, 2F, 5F |
| 2 | TestLoop(90, new int[] 15); | Path 2: 1T, 2T, 3T, 2F, 5F |
| 3 | TestLoop(90, new int[] 15, 0); | Path 3: 1T, 2T, 3T, 2T, 3F, 2F, 5F |
| 4 | TestLoop(90, new int[] 15, 15); | Path 4: 1T, 2T, 3T, 2T, 3T, 2F, 5F |

Form 1. Several explored paths

Their fitness values for these paths are NA (for that the target test is not even reached), 20($|110 - 90|$), 19($|110 - 91|$), 19($|110 - 91|$), and 18(—110 - 92—) respectively. According to the proposed algorithm, the path 4 will be prefered since it the smallest value among all the paths which means that Path 4 is the closest path to the target test. Using symbolic value to compare fitness values may not to be a good idea since the comparison between symbolic values is quite expensive due to the invoking of constraint solver and the pairwise comparison is needed. Therefore, the concrete values are computed and compared. In general, the predicate can be used to generate fitness functions to calculate how close a execution path to target test. The follwing form shows several instances, column 1 shows the predicate of the target test, column 2 shows the fitness function derived from it.

| Predicate | Fitness function |
|---|---|
| $F(a == b)$ | $\|a - b\|$ |
| $F(a > b)$ | $(b - a) + k$ |
| $F(a >= b)$ | $(b - a)$ |
| $F(a < b)$ | $(a - b) + k$ |
| $F(a <= b)$ | $(a - b)$ |

Form 2. Predicates and fitness function.

Besides the kind of predicates listed above, there is another category of target test that are guarded with boolean value rather than number value. In that case, deriving the fitness function will be much harder and no strategy has been proposed.

**Fitness-gain computation for a branch.** Selecting branch node in a path to flip in order to approach the target test could be reduced to selecting a branch node that (1)has at least one new branch that has not been covered yet,(2)has the best potential to improve the fitness value of the current path. The fitness gain is defined as the decreasement of the fitness value before and after the flipping of a branch. The fitness gain could be used to prioritize the flipping of various branches. For example, the fitness-gain of flipping the branch in line 3 for path 1 is 1; The flipping of same branch in Path 3 will result in fitness-gain of same degree. Intuitively, not all the flipping will result in desirable fitness-gain. For example, the flip from true to false of line 3 in path 3 will cause the fitness-gain of -1.

Each branch b in a path p is assigned a composite number $F(p) - FGain(b)$, whare $F(p)$ is the fitness value of the path p, and $FGain(b)$ is the fitness-gain of the flipping b in p. Then the priority of the flips all over the explored paths could be decided according to the composite value. Consider the previous four tests as a example, the smallest composite value comes at flipping the false branch in line 2 of the path 4 since the node has the least composite value 17. For several iterations, the strategy will eventually achieve the path whose fitness value is exactly 0, in another word, reach the target test.

## 4 Selective Symbolic Execution

Symbolic execution [16] is used to reason path-flow behavior of a program by tracing path constraint information collected during symbolic execution. This technique is useful in automated software testing such as increasing path coverage.

There exist two issues with regards to limited scalability of symbolic execution. The first issue is that this technique can be applied on only small program with at most thousands of lines of code. Note that symbolic execution collects states and constraints to keep track of path information in a program. During symbolic execution, the size of such states and constraints is exponential to the number of conditions (e.g., as if-else statements in program) in a program. This problem is called as path explosion. Due to such an exponential growth, the technique is not yet applied to large programs (e.g., program with more than millions lines of code).

The second issue is related to symbolic execution interacting with environments such as library or third-party program. Many programs are often interacting with environments such as external libraries to control network devices. For example, if a program under test uses external libraries, the program may require symbolic execution on external libraries as well. Such symbolic execution of environments could result in path explosion.

Ideally, testers often try to achieve covering all or specific paths of a system. However, in practice, covering such paths is not trivial due to limited scalability caused by path explosion. To address the issue, Chipounov et al. [9] propose selective symbolic execution technique. The main idea of this technique, called selective symbolic execution ($S^2E$), combines both symbolic execution and concrete execution. More specifically, users specify portions (in a system) of interest. For example, portions of interest could be program related to CPU or memory usage to detect deadlock or memory leakage. $S^2E$ explores such portions (of interest) symbolically (i.e., in-scope executions) and the remaining portions (of non-interest) concretely (i.e., out-scope executions). Therefore, execution of a program under test is back and forth between the symbolic and concrete executions based on which portions are executed.

$S^2E$ can be practical since the authors observe that developers often focus on only small portion of code in a system (e.g., kernel module or newly added feature) instead of testing a whole system. S2E can be useful for many important software development tasks as follows.

Selective symbolic execution ($S^2E$) has two dimensions in a system; code and data. Users can specify portions of code of interest by indicating file name or range of program counters. Given code information, $S^2E$ executes the portion of code symbolically. All variables in the portion in conditional branches are involved in symbolic execution. Therefore, all paths could be explored symbolically. Users can specify portions of data of interest by indicating a data structure's name or address range of data segment. Then, $S^2E$ executes all portion of code related to the data of interest symbolically. Figure 6 shows an example of $S^2E$ execution. In the example, the main function calls functions $f$ and $f_2$. Consider that users specify $f$ is portion (of interest) to be executed symbolically. To achieve the goal, when the main function calls the $f$ that has an input parameter $X$, $X$ is assigned to a symbolic value, $\lambda$ and $f$ is symbolically executed. However, when the main function calls the $f_2$ of non-interest, $S^2E$ assigns $X$ as "3" to execute $f_2$ concretely.

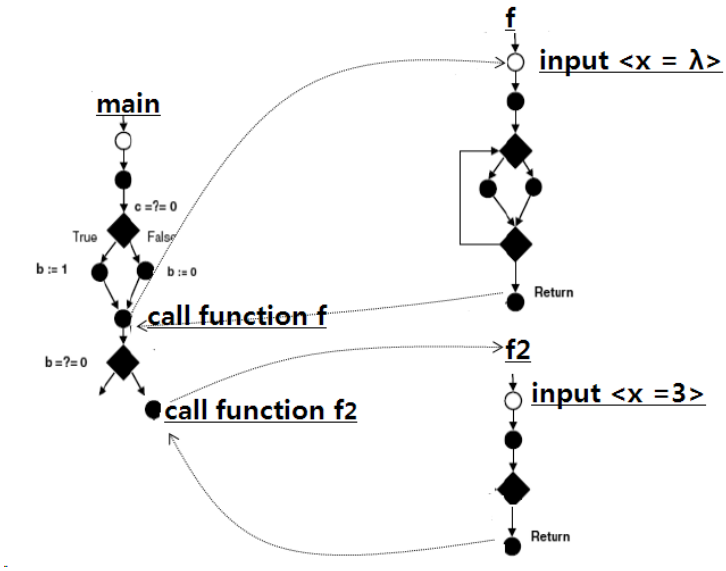


**Figure 6. Example of selective symbolic execution**

## 5 Major Contributors

The concept of symbolic execution was introduced academically with descriptions of the Select system, proposed by Boyer et al [6]. In 1976, test data generation using symbolic execution was first proposed by James C. King [16]. Around the same time, Clarke also did the work on this [10]. Their pioneer works open the ways to automatic test data generation by using symbolic execution to do static analysis of code for path safety and prove theorems about code. However, these static ways faced the exponential state space explosion problem, which made it only practical for small programs. In recent years, Koushik Sen, whose paper on concolic testing [13] won the ACM SIGSOFT Distinguished Paper Award at ESEC/FSE '05, proposed CUTE and DART tools, blossoming up the path-based automatic test data generation using symbolic execution. With the impressive progress of constraint solvers and concolic path-based testing [25, 4, 7, 8, 23, 29], it is possible to perform automatic path-based testing on large scale programs. Pex [29], a symbolic execution test generation tool for .NET proposed by Nikolai Tillmann, has been used to test .NET core libraries and found serious bugs. To alleviate the classic path explosion problem, many new techniques are also been proposed by Xie [30], Godefroid [12] and so on.

## References

[1] B. Botella A. Gotlieb and M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In *ISSTA*, 1998.

[2] B. Botella A. Gotlieb and M. Watel. Inka: Ten years after the first ideas. In *ICSSEA*, 2006.

[3] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, Cambridge, UK, 2008.

[4] S. Bardin and P. Herrmann. Structural Testing of Executables. In *ICST*, 2008.

[5] Sébastien Bardin and Philippe Herrmann. Pruning the Search Space in Path-Based Test Generation. In *ICST*, 2009.

[6] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT–a formal system for testing and debugging programs by symbolic execution. In *ICRS*, 1975.

[7] Cristian Cadar and Dawson Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *SPIN*, 2005.

[8] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *CCS*, 2006.

[9] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective Symbolic Execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[10] Lori A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *TSE*, 1976.

[11] V. Kahlon C.Wang, Z. Yang and A. Gupta. Peephole Partial Order Reduction. In *TACAS*, 2008.

[12] Patrice Godefroid. Compositional Dynamic Test Generation. In *POPL*, 2007.

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[14] Jonathan Gross and Jay Yellen. *Graph theory and its applications*. CRC Press, Inc., Boca Raton, FL, USA, 1999.

[15] M. Harman and P. McMinn. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *ISSTA*, 2007.

[16] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[17] B. Korel. Automated Software Test Data Generation. In *TSE*, 1990.

[18] B. Korel. Automated Test Data Generation for Programs with Procedures. In *ISSTA*, 1996.

[19] R. Majumdar and K. Sen. Hybrid Concolic Testing. In *ICSE*, 2007.

[20] A. P. Mathur N. Gupta and M. L. Soffa. Automated Test Data Generation Using an Iterative Relaxation Method. In *FSE*, 1998.

[21] A. P. Mathur andM. L. Soffa. N. Gupta. UNA Based Iterative Test Data Generation and its Evaluation. In *ASE*, 1999.

[22] B. Marre N. Williams and P. Mouy. On-the-Fly Generation of K-Path Tests for C Functions. In *ASE*, 2004.

[23] M. Y. Levin P. Godefroid and D.Molnar. Automated Whitebox Fuzz Testing. In *NDSS*, 2008.

[24] N.Williams P.Mouy, B.Marre and P. Le Gall. Generation of All-Paths Unit Test with Function Calls. In *ICST*, 2008.

[25] C. S. Pasareanu S. Anand and W. Visser. JPF-SE: A Symbolic Execution Extension to Java PathFinder. In *TACAS*, 2007.

[26] P. Godefroid S. Anand and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *TACAS*, 2008.

[27] K. Sen and G. Agha. Automated Systematic Testing of Open Distributed Programs. In *FASE*, 2006.

[28] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE*, 2005.

[29] N. Tillmann and J. de Halleux. Pex-White Box Test Generation for .NET. In *TAP*, 2008.

[30] Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN*, June-July 2009.