Symbolic Execution for Software Testing in Practice – Preliminary Assessment

Cristian Cadar
Imperial College London
c.cadar@imperial.ac.uk

Patrice Godefroid
Microsoft
pg@microsoft.com

Sarfraz Khurshid
U. Texas at Austin
khurshid@ece.utexas.edu

Corina Păsăreanu[*]
NASA Ames
corina.s.pasareanu@nasa.gov

Koushik Sen
U.C. Berkeley
ksen@eecs.berkeley.edu

Nikolai Tillmann
Microsoft
nikolait@microsoft.com

Willem Visser
U. Stellenbosch
willem@gmail.com

ABSTRACT

We present results for the "Impact Project Focus Area" on the topic of symbolic execution as used in software testing. Symbolic execution is a program analysis technique introduced in the 70s that has received renewed interest in recent years, due to algorithmic advances and increased availability of computational power and constraint solving technology. We review classical symbolic execution and some modern extensions such as generalized symbolic execution and dynamic testing. We also give a preliminary assessment of the use in academia, research labs, and industry.

1. INTRODUCTION

The ACM-SIGSOFT Impact Project is documenting the impact that software engineering research has had on software development practice. In this paper, we present preliminary results for documenting the impact of research in symbolic execution for automated software testing. Symbolic execution is a program analysis technique that was introduced in the 70s [9, 16, 26, 30, 39], and that has found renewed interest in recent years [13, 14, 24, 28, 42–44], followed by many other works [10, 25, 27, 35, 37, 38, 46, 47] etc.

Symbolic execution is now the underlying technique of several popular testing tools, many of them open-source: NASA's Symbolic (Java) PathFinder[1], UIUC's CUTE and jCUTE[2], Stanford's KLEE[3], UC Berkeley's CREST[4] and BitBlaze[5], etc. Symbolic execution tools are now used in industrial practice at Microsoft (PREfix [11], Pex[6], SAGE [25] and YOGI[7]), IBM (Apollo [2]), NASA and Fujitsu (Symbolic PathFinder), and also form a key part of the commercial testing tool-suites from Parasoft and other companies [50].

Although we acknowledge that the impact of symbolic execution in software practice is still limited, we believe that the explosion of work in this area over the past years makes for an interesting story about the increasing impact of symbolic execution since it was first introduced in the 1970s.

Software testing is the most commonly used technique for validating the quality of software, but it is typically a mostly manual process that accounts for half of the total cost of software development and maintenance [6]. Symbolic execution is one of the many techniques that can be used to automate software testing. In particular, symbolic execution can be used to automatically generate test cases that achieve high code coverage. Symbolic execution is a program analysis technique that executes programs with symbolic rather than concrete inputs and maintains a *path condition* that is updated whenever a branch instruction is executed, to encode the constraints on the inputs that reach that program point. Test generation is performed by solving the collected constraints, using a decision procedure or constraint solver. Symbolic execution can also be used for bug finding, where it checks for run-time errors or assertion violations and it generates test inputs that trigger those errors.

The original approaches to symbolic execution [9, 16, 26, 30, 39] addressed simple sequential programs with a fixed number of input data of primitive type. Modern approaches, such as generalized symbolic execution (GSE) [28] and jCUTE [43], address multi-threaded programs with complex data structures as inputs. Much of the popularity of symbolic execution is due to dynamic or concolic testing [24, 44], a variant of *dynamic symbolic execution* [31], where the symbolic execution is performed at run-time, along concrete program executions. A closely related approach is advocated by Execution Generated Testing (EXE) [13], which performs mixed concrete/symbolic execution. We discuss these techniques in more detail in the next section.

Symbolic execution still suffers from scalability issues due to the large number of paths that need to be analyzed and the

[*]We thank Matt Dwyer for his advice
[1]http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
[2]http://osl.cs.uiuc.edu/~ksen/cute/
[3]http://klee.llvm.org/
[4]http://code.google.com/p/crest/
[5]http://bitblaze.cs.berkeley.edu/
[6]http://research.microsoft.com/en-us/projects/pex/
[7]http://research.microsoft.com/en-us/projects/yogi/

ICSE 2011 Waikiki, Honolulu, Hawaii USA

complexity of the constraints that get generated (see also Section 4). However, algorithmic advances, newly available SMT solvers[8] and more powerful computers have already made it possible to apply such techniques to large programs (with millions lines of code) and to discover subtle bugs in commonly used software – ranging from library code to network and operating systems code – saving millions of dollars (see Section 3).

2. SYMBOLIC EXECUTION

2.1 "Classical" Symbolic Execution

The key idea behind symbolic execution [30] is to use *symbolic values*, instead of actual data, as input values, and to represent the values of program variables as symbolic expressions. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs.

Symbolic execution maintains a symbolic state, which maps variables to symbolic expressions, and a symbolic path constraint PC, a first order quantifier free formula over symbolic expressions. PC accumulates constraints on the inputs that trigger the execution to follow the associated path. At every conditional statement `if (`e`) S1 else S2`, PC is updated with conditions on the inputs to choose between alternative paths. A fresh path constraint PC' is created and updated to $PC \wedge \neg\sigma(e)$ ("else" branch) and PC is updated to $PC \wedge \sigma(e)$ ("then" branch). Note that unlike in concrete execution, both branches can be taken, resulting in two execution paths. If any of PC or PC' becomes unsatisfiable, symbolic execution terminates along the corresponding path. Satisfiability is checked with an off-the-shelf constraint solver.

Whenever symbolic execution along a path terminates (normally or with an error), the current PC is solved and the solution forms the *test inputs*—if the program is executed on these concrete inputs, it will take the same path as the symbolic execution and terminate. Symbolic execution of code containing loops or recursion may result in an infinite number of paths; therefore, in practice, one needs to put a limit on the search (e.g., a timeout or a limit on the number of paths or exploration depth).

2.2 Generalized Symbolic Execution

Generalized symbolic execution (GSE) [28] extends classical symbolic execution with the ability of handling multi-threading and input recursive data structures. GSE implements symbolic execution on top of a standard model checker, whose built-in capabilities are used for handling multi-threading (and other forms of non-determinism). GSE handles input recursive data structures by using *lazy initialization.* GSE starts execution of the method on inputs with *uninitialized* fields and initializes fields when they are first accessed during the method's symbolic execution. This allows symbolic execution of methods without requiring an a priori bound on the number of input objects. Method preconditions can be used to ensure that fields are initialized to values permitted by the precondition. Partial correctness properties are given as assertions in the program.

[8] http://www.smtcomp.org/

Lazy Initialization: On the first access to an un-initialized reference field, GSE non-deterministically initializes it to `null`, to a reference to a new object with un-initialized fields or to a reference to an object created during a prior initialization step; this systematically treats aliasing. Once the field has been initialized, the execution proceeds according to the concrete execution semantics. The model-checker is used to handle the non-determinism introduced when creating different heap configurations and when updating path conditions.

2.3 Directed Automated Random Testing

Directed Automated Random Testing [24] (also known as DART, concolic testing [44], or dynamic symbolic execution [47]) performs symbolic execution while the program is executed on some concrete input values. DART maintains a concrete state and a symbolic state simultaneously. DART executes a program starting with some given or random input, gathers symbolic constraints on inputs at conditional statements along the execution, and then uses a constraint solver to infer variants of the previous inputs in order to steer the next execution of the program towards an alternative feasible execution path. This process is repeated systematically or heuristically until all feasible execution paths are explored or a user-defined coverage criteria is met.

A key observation in DART is that *imprecision in symbolic execution can be alleviated using concrete values and randomization*: whenever symbolic execution does not know how to generate a constraint for a program statement depending on some inputs, one can always simplify this constraint using the concrete run-time values of those inputs. In those cases, symbolic execution degrades gracefully by leveraging concrete values into a form of partial symbolic execution.

CUTE (A Concolic Unit Testing Engine) and jCUTE (CUTE for Java) [42–44] extends DART to handle multi-threaded programs that manipulates dynamic data structures using pointer operations. CUTE avoids imprecision due to pointer analysis by representing and solving *pointer constraints* approximately. In multi-threaded programs, CUTE combines concolic execution with dynamic partial order reduction to systematically generate both test inputs and thread schedules.

2.4 Execution Generated Testing

Execution generated test cases (EXE) [13,14] optimizes symbolic execution by making a distinction between the concrete and the symbolic state of a program (i.e., by performing *mixed concrete/symbolic execution*). A key advantage of this approach is that the concrete state does not need to be explicitly maintained—it is maintained as part of the normal execution state of the program.

EXE is designed for comprehensively testing complex software, with an emphasis on systems code. EXE works on C code, and can build constraints for all C expressions with perfect *bit-level accuracy*, including those involving pointers, casting, unions, and bit-fields. (The main exception is floating-point, which the current generation of constraint solvers does not handle.) As importantly, EXE provides the speed necessary to quickly solve many of these constraints, through a combination of low-level optimizations implemented in its purposely designed constraint solver STP [14,21], and a series of higher-level ones such as caching and irrelevant constraint elimination.

To deal with the complexities of systems code, EXE models memory with *bit-level accuracy.* This is because systems code often treats memory as untyped bytes, and observes a single memory location in multiple ways: e.g., by casting signed variables to unsigned, or treating an array of bytes as a network packet, inode, or packet filter through pointer casting.

3. TOOLS AND IMPACT

In this section we present several recent tools that are based on symbolic execution, together with a preliminary assessment of their impact in practice. In the very limited scope of this paper, it is impossible to review here all the relevant tools. Instead, we focus on a few representative ones that implement the different flavors of symbolic execution presented in the previous section. Albeit incomplete, we do hope that this list convinces the reader of the growing impact of symbolic execution in practice.

JPF–SE and Symbolic (Java) PathFinder. The original GSE framework was developed for Java programs and used NASA's Java PathFinder (JPF) model checker as an enabling technology (see JPF–SE [1]), although GSE can be made to work with other model checkers and imperative languages. Since JPF is a general purpose model checker, GSE benefits from its collection of built-in state space exploration capabilities, such as different search strategies (e.g., heuristic search) as well as partial order and symmetry reductions; (abstract) state matching can be used to avoid performing redundant work [49]. A similar tool [20] uses the Bogor model checking framework, instead of JPF, while yet another approach uses SPIN [45], for checking parallel numeric applications.

SPF can analyze both *Java bytecode* and statechart *models*, e.g., Simulink/Stateflow, Standard UML, Rhapsody UML, etc., via automatic translation into bytecode. SPF can handle mixed integer and real constraints, and complex mathematical constraints, via heuristic solving. A parallel version also exists [46].

SPF is part of the JPF project[9] (open-sourced since 2003) and it has been applied at NASA in various projects, such as test case generation for the Orion control software (where it helped uncover subtle bugs [38]), fault tolerant protocols, NextGen (TSAFE) aviation software or robot executives. SPF has been extended with a symbolic string analysis at Fujitsu, where it is being used for testing web applications[10]. MIT's tool JFuzz [27], a concolic whitebox fuzzer for Java, has been built on top of SPF and it is freely available from the JPF web site.

DART. The DART tool implements Directed Automated Random Testing [24], which was presented in Section 2.3. DART aims at *systematically* executing *all* (or as many as possible) feasible paths of a program, while checking each execution for various types of errors. In practice, such a directed search typically cannot explore all the feasible paths of large programs in a reasonable amount of time, but it usually does achieve much better coverage than pure random testing and, hence, find new program bugs.

DART was originally developed at Bell Labs and targeted C code. This style of systematic dynamic test generation has been implemented in many other tools over the last few years, including Pex [47], SAGE [25], CREST [10] and SPLAT [36].

CUTE and jCUTE. CUTE and jCUTE were developed in University of Illinois at Urbana-Champaign for C and Java programs, respectively. They implement concolic testing and handle input data structures and multi-threading. Both tools have been applied to test several open-source software including `java.util` library of Sun' JDK 1.4 and bugs detected by these tools have been made available to the developers. Concolic testing has also been studied in different courses at several universities including Stanford, Berkeley, UIUC, CMU, Georgia Tech, Purdue, UC Santa Barbara, North Carolina State University, UC Santa Cruz, UC San Diego.

CREST. CREST [10] is an open-source tool for concolic testing of C programs. CREST is an extensible platform for building and experimenting with heuristics for selecting which paths to test for programs with far too many executions paths to exhaustively explore. Since being released as an open source in May 2008 [11], CREST has been downloaded 1500+ times and has been used by several research groups. For example, CREST has been used to build tools for augmenting existing test suites to test newly-changed code [52] and for detecting SQL injection vulnerabilities [40], has been modified to run distributed on a cluster for testing a flash storage platform [29], and has been used to experiment with more sophisticated concolic search heuristics [4]. CREST has also been used in teaching courses at few universities.

SAGE: Automated Whitebox Fuzzing Whitebox fuzzing [25] is a recent approach to security testing which extends the scope of systematic dynamic test generation from unit testing to whole-application testing. Whitebox fuzzing is able to scale to large file parsers embedded in applications with millions of lines of code and execution traces with billions of machine instructions, such as Microsoft Excel. Several key technical innovations made this possible: new techniques for symbolically executing very long execution traces with billions of program instructions, for symbolic execution at the x86 assembly level, for compact representation of path constraints, new embarassingly-parallel state-space search algorithms like the generational search, and new support in SMT solvers (such as Z3 [19]) for test generation.

Whitebox fuzzing was first implemented in SAGE [25] and since adopted in several other tools, such as CatchConv, Fuzzgrind, Immunity, etc. Over the last couple of years, whitebox fuzzers have found many new security vulnerabili-

[9] http://babelfish.arc.nasa.gov/trac/jpf

[10] http://www.fujitsu.com/global/news/pr/archives/month/2010/20100112-02.html

[11] Available at http://code.google.com/p/crest

ties (buffer overflows) in Windows [25] and Linux [37] applications, including codecs, image viewers and media players. Notably, SAGE found roughly one third of *all* the bugs discovered by file fuzzing during the development of Microsoft's Windows 7 [23], saving millions of dollars by avoiding expensive security patches for nearly a billion PCs worldwide. Since 2008, SAGE has been continually running on an average of 100+ machines automatically "fuzzing" hundreds of applications in a dedicated security testing lab. To date, this represents the largest computational usage ever for any SMT solver, according to the authors of the Z3 SMT solver [19].

Pex. Pex [47] implements Dynamic Symbolic Execution to generate test inputs for .NET code, supporting languages such as C#, VisualBasic, and F#. Pex extends the basic approach in several unique ways: While Pex can use concrete values to simplify constraints, Pex usually faithfully represents the semantics of almost all .NET instructions symbolically, including safe and unsafe code, as well as instructions that refer to the object oriented .NET type system, such as type tests and virtual method invocations. Pex uses the SMT solver Z3 [19] to compute models, i.e. test inputs, for satisfiable constraint systems. Pex uses approximations for theories for which Z3 has no precise decision procedures, e.g. for string [7] and floating point arithmetic [33]. Pex supports the generation of test inputs of primitive types as well as (recursive) complex data types, for which Pex automatically computes a factory method which creates an instance of a complex data type by invoking a constructor and a sequence of methods, whose parameters are also determined by Pex. Pex combines several search strategies which select the order in which different execution paths are attempted, in order to achieve high code coverage quickly [51]. In addition to the test case generation capabilities, Pex comes with a mock and stub framework, which makes it easy to write and reuse models for .NET libraries. [18]. Pex enables Parameterized Unit Testing [48], an extension of traditional unit testing.

Pex is a Visual Studio 2010 Power Tool[12]. It is used by several groups within Microsoft. Externally, Pex is available under academic and commercial licenses. The stand-alone Pex tool has been downloaded more than 40,000 times. Anyone can try out Pex in the browser[13], where visitors let Pex analyze more than 250,000 programs within the first five months of the launch of the website.

KLEE. KLEE is a complete redesign of the EXE tool [13, 14], built on top of the LLVM [34] compiler infrastructure. Like EXE, it performs mixed concrete/symbolic execution, models memory with bit-level accuracy, employs a variety of constraint solving optimizations, and uses search heuristics to get high code coverage.

One of the key improvements of KLEE over EXE is its ability to store a much larger number of concurrent states, by exploiting sharing among states at the object-, rather than at the page-level as in EXE. Another important improvement is its ability to handle interactions with the outside *environment* — e.g., with data read from the file system or over the network — by providing environment models, whose goal is to explore all possible legal interactions with the outside world.

[12] `http://msdn.microsoft.com/en-us/vstudio/bb980963.aspx`

[13] `http://pexforfun.com`

As a result of these features, KLEE and its predecessor EXE were capable to automatically generate high-coverage test suites, and to discover deep bugs and security vulnerabilities in a variety of complex code, ranging from library code to UNIX utilities, file systems, packet filters, device drivers and network servers and tools [8, 12–14, 53].

KLEE has been open-sourced in June 2009[14]. Since then, KLEE has been downloaded by a variety of groups from both the academia and the industry. In particular, it has been used and extended by several research groups, with some of these extensions being contributed back to the main branch. These users have applied KLEE to a variety of areas, ranging from wireless sensor networks [41] to automated debugging [54], reverse engineering and testing of binary device drivers [15,32], exploit generation [3], online gaming [5], and schedule memoization in multithreaded code [17].

4. CONCLUSION

In this paper we have focused on modern symbolic execution technique that have become popular in recent years; we have also reviewed the associated tools and their impact in practice. We outline here some of the challenges to symbolic execution and its wider adoption in software engineering practice.

A significant scalability challenge for symbolic execution is how to handle the exponential number of paths in the code. Significant advances in compositional techniques [22], pruning redundant paths [8], and heuristics search [10, 35] are needed. Parallelization should also help [46], since the paths generated by symbolic execution can be analyzed independently.

Real applications often require solving complex, non-linear mathematical constraints that are undecidable or very hard to solve; new heuristic techniques are necessary to solve such problems. Test case generation for web applications and security problems requires solving string constraints and combinations of numeric and string constraints. Progress in these areas would significantly extend the impact of symbolic execution to new application domains.

5. REFERENCES

[1] S. Anand, C. S. Păsăreanu, and W. Visser. Jpf-se: a symbolic execution extension to java pathfinder. In *TACAS'07*, pages 134–138, 2007.

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *ISSTA'08*, July 2008.

[3] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In *NDSS'11*, Feb 2011.

[4] M. Baluda, P. Braione, G. Denaro, and M. Pezzè. Structural coverage of feasible code. In *AST'10*, pages 59–66. ACM, 2010.

[5] D. Bethea, R. Cochran, and M. Reiter. Server-side verification of client behavior in online games. In *NDSS'10*, Feb–Mar 2010.

[14] KLEE is available at `http://klee.llvm.org`.

[6] B. Bezier. *Software Testing Techniques, 2nd Edition.* Van Nostrand Reinhold, New York, 1990.

[7] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS'09*, Mar. 2009.

[8] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS'08*, Mar–Apr 2008.

[9] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10:234–245, Apr 1975.

[10] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE'08*, 2008.

[11] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software–Practice and Experience*, 30(7):775–802, 2000.

[12] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*, Dec 2008.

[13] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself (invited paper). In *SPIN'05*, Aug 2005.

[14] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *CCS'06*, Oct–Nov 2006.

[15] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *EuroSys'10*, Apr 2010.

[16] L. A. Clarke. A program testing system. In *Proc. of the 1976 annual conference*, pages 488–491, 1976.

[17] H. Cui, J. Wu, C. che Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *OSDI'10*, 2010.

[18] J. de Halleux and N. Tillmann. Moles: Tool-assisted environment isolation with closures. In *TOOLS'10*, June–July 2010.

[19] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *TACAS'08*, Mar–Apr 2008.

[20] X. Deng, J. Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *ASE'06*, 2006.

[21] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV'07*, July 2007.

[22] P. Godefroid. Compositional dynamic test generation. In *Proc. of the ACM POPL*, Jan 2007.

[23] P. Godefroid. Software model checking improving security of a billion computers. In *SPIN*, page 1, 2009.

[24] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI'05*, 2005.

[25] P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS'08*, Feb. 2008.

[26] W. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.

[27] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jFuzz: A concolic whitebox fuzzer for java. In *In NFM'09*, 2009.

[28] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS'03*, Apr. 2003.

[29] Y. Kim, M. Kim, and N. Dang. Scalable distributed concolic testing: a case study on a flash storage platform. In *ICTAC'10*, pages 199–213, 2010.

[30] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[31] B. Korel. A Dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, Nov 1990.

[32] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX ATC'10*, June 2010.

[33] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux. Flopsy - search-based floating point constraint solving for symbolic execution. In *ICTSS'10*, pages 142–157, 2010.

[34] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO'04*, Mar 2004.

[35] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE'07*, May 2007.

[36] R. Majumdar and R.-G. Xu. Reducing test inputs using information partitions. In *CAV'09*, pages 555–569, 2009.

[37] D. Molnar, X. C. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security'09*, Aug 2009.

[38] C. Pasareanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA'08*, July 2008.

[39] C. Ramamoorthy, S.-B. Ho, and W. Chen. On the automated generation of program test data. *IEEE Trans. on Software Engineering*, 2(4):293–300, 1976.

[40] M. Ruse, T. Sarkar, and S. Basu. Analysis & detection of sql injection vulnerabilities via automatic test case generation of programs. *IEEE/IPSJ Int. Sym. Applications and the Internet*, 2010.

[41] R. Sasnauskas, J. A. B. Link, M. H. Alizai, and K. Wehrle. Kleenet: automatic bug hunting in sensor network applications. In *IPSN'10*, Apr 2010.

[42] K. Sen. *Scalable Automated Methods for Dynamic Program Analysis.* PhD thesis, University of Illinois at Urbana-Champaign, June 2006.

[43] K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *CAV'06*, 2006.

[44] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE'05*, Sep 2005.

[45] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM TOSEM*, 17:10:1–10:34, May 2008.

[46] M. Staats and C. S. Pasareanu. Parallel symbolic execution for structural test generation. In *ISSTA'10*, July 2010.

[47] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *TAP'08*, Apr 2008.

[48] N. Tillmann and W. Schulte. Parameterized unit tests. In *ESEC/FSE'05*, Sept. 2005.

[49] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for java containers using state matching. In *ISSTA'06*. ACM, 2006.

[50] T. Xie. Improving automation in developer testing: State of practice. Technical report, North Carolina State University, 2009.

[51] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN'09*, June-July 2009.

[52] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *FSE'10*, FSE '10. ACM, 2010.

[53] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *IEEE Symposium on Security and Privacy*, May 2006.

[54] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In

EuroSys'10, Apr 2010.