FSX: A Tool for Fine-Grained Incremental Unit Test Generation for C/C++ Programs

Hiroaki Yoshida†, Susumu Tokumoto‡, Mukul R. Prasad†, Indradeep Ghosh†, Tadahiro Uehara‡
† Fujitsu Laboratories of America, Inc., USA
‡ Fujitsu Laboratories Ltd., Japan

ABSTRACT

Automated unit test generation bears the promise of significantly reducing test cost and hence improving software quality. However, the maintenance cost of the automatically generated tests presents a significant barrier to adoption of this technology. To address this challenge, in previous work, we proposed a novel technique for automated and fine-grained incremental generation of unit tests through minimal augmentation of an existing test suite. In this paper we describe a tool FSX, implementing this technique. We describe the architecture, user-interface, and salient features of FSX, and specific practical use-cases of its technology. We also report on a real, large-scale deployment of FSX as a practical validation of the underlying research contribution and of automated test generation research in general.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Automatic test generation; symbolic execution; unit testing

1. INTRODUCTION

Testing is the dominant process for establishing confidence in the correctness of software [21]. Functional unit tests, that test individual functions, are a key component of software testing. However, high-quality unit test suites are notoriously laborious to develop [12]. Automatic test generation, which has been actively researched over the past two decades, can help reduce the cost of software testing [26, 19, 33, 37, 17, 14, 22].

A significant barrier to the practical adoption of automatic (*unit*) test generation tools is the maintainability cost of the generated tests to the human developers responsible for them. The importance of writing compact test cases is well recognized as a means of improving maintainability of tests. For example the GNU bug reporting instructions [3] espouse that, "smaller test cases make debugging easier", "GCC developers prefer bug reports with small, portable test cases" and "minimized test cases can be added to the GCC test suites". Other practical software projects, such as LLVM [5], Mozilla [8], and Webkit [11], mirror this view on compact, minimal (unit) tests.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2983937

In [41] we presented an approach and software prototype FSX that approaches (unit) test generation from the standpoint of the maintenance cost of generated test code. FSX is built on the premise that every new line of test code adds to the maintenance cost of the test suite - the developer needs to understand and maintain that line. Thus, FSX attempts to minimize the number of lines of newly generated test code. Existing approaches for test-suite augmentation [31, 38, 35, 28, 40, 23], *i.e.,* using the test-suite for a previous revision of the program to build the test-suite for the current revision, operate at the granularity of *complete* test cases, *i.e.,* complete new tests are added to the test-suite. The existing test suite is primarily used to identify test targets (*e.g.,* branches, lines, or execution paths) not covered by current tests. By contrast, FSX treats each part of existing cases, including the test driver, test input data, and oracles, as "test intelligence". It attempts to create tests for uncovered test targets by copying and minimally modifying existing tests, where possible, rather than creating new ones.

The FSX approach is built on two key and novel pieces of technology. The first, is a technique for iterative, incremental refinement of test-cases by concurrent driver generation and symbolic execution. Instead of making all variables symbolic, it starts with a minimal test driver (or an existing test driver when available) with concrete input values. Then, based on diagnostic information obtained during symbolic execution, the driver, and test-case is progressively enhanced by making only relevant variables symbolic, or assigning appropriate objects. The second, is a precise and efficient byte-level dynamic dependence analysis, based on Reduced Ordered Binary Decision Diagrams (ROBDDs) [13]. This analysis tracks the correspondence between input values and symbolic expressions, generating precise diagnostic information to guide the (minimal) iterative enhancement of test drivers.

This paper addresses the implementation and deployment of the FSX tool. Specifically, it makes the following contributions:

- It provides a detailed description of the architecture and user interface of FSX as well as the software engineering practices used to develop and maintain it (Section 4).
- It describes two specific use-cases of FSX (Section 3).
- It reports on a real, large-scale deployment of FSX (Section 6). We see this deployment as a practical validation of not only FSX's technology but in fact the larger body of work in automated test generation that it builds upon.

2. TECHNIQUE

FSX's fine-grained incremental test generation method consists of three main components: *Incremental Symbolic Execution*, *Event Diagnosis*, and *Incremental Driver Generation*, as shown in Figure 1. Given a function-under-test and a set of previous tests [1], it generates an updated set of tests, with the objective of maximizing the structural coverage of the generated tests, while minimizing the difference between the generated and previous tests.

[1]Test drivers are simply tests with some of the inputs symbolized.

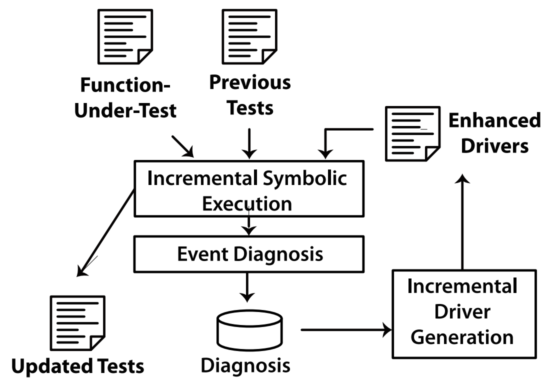

Figure 1: Overview of incremental test generation in FSX.

To perform test generation, FSX's incremental symbolic execution engine executes the drivers one at a time. If the test terminates without any *indicative events*, it is retained in the new test suite. Indicative events are either exceptions, such as null pointer or out-of-bound memory accesses, or the discovery of uncovered test targets (*e.g.,* branches). Intuitively, indicative events constitute targets for subsequent test generation. Any indicative events are passed on to the event diagnosis engine which generates diagnosis information explaining the causes of each event. Given the set of diagnoses, incremental driver generation enhances the previous drivers by attempting to resolve the indicative events. Then, the entire process, starting with incremental symbolic execution, is iterated on the newly obtained enhanced drivers until the coverage goal is satisfied or the drivers cannot be enhanced any further.

Incremental symbolic execution. Conceptually, FSX's incremental symbolic execution engine can be seen as a standard symbolic executor for C programs, such as KLEE [14], enhanced with symbolic support for C++-specific language constructs, intrinsics and libraries, such as in KLOVER [25], and further enhanced with path-pruning techniques based on recorded previous execution history, similar to DISE [28] or Memoise [40]. In addition to standard test generation, it also collects indicative events, which it passes on to the event diagnosis engine. Note, however, that the FSX tool is built from scratch. It does not share code with any of these tools.

Event diagnosis. The event diagnosis engine outputs a diagnosis, *i.e.,* a set of causes, for a given indicative event. To do this it collects several pieces of diagnostic information during the symbolic computation, principally, 1) for each computed value, the set of inputs, called the *relevant input set*, from which the value is derived, 2) the type and structure of untyped input data, dynamically cast to specific types, within the function-under-test, as it is used, and 3) range information for composite types, such as arrays (to facilitate fixing of out-of-bound memory access events). One of the key innovations of FSX is the precise computation of relevant input sets, which are encoded as Boolean characteristic functions, and compactly represented through Reduced-Ordered Binary Decision Diagrams (ROBDDs) [13].

Incremental driver generation. Incremental driver generation enhances the previous drivers, to eliminate any indicative events, by suitably modifying the values of inputs relevant to such events. This is done based on the recorded diagnostic information, including the relevant input sets, and the type of the event. For a null pointer access the corresponding pointer is assigned to either a new object allocated based on its type information or a pointer to an already-allocated object compatible to its type. For an out-of-bound memory access, the accessed object is resized to be sufficiently bigger, based on recorded range information. In case of a branch-not-taken event, the relevant inputs are assigned symbolic values using one of several strategies implemented in FSX.

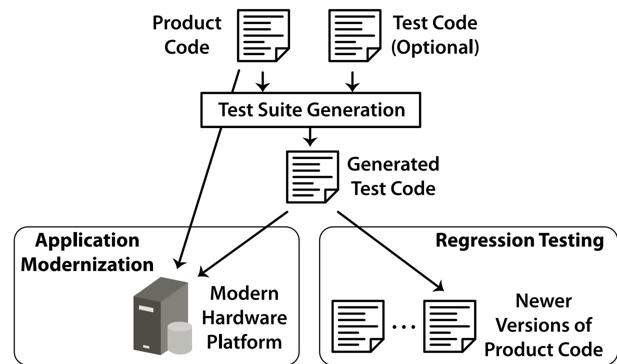

Figure 2: Application modernization and regression testing.

3. USAGE SCENARIOS

3.1 Application Modernization & Regression Testing

Application modernization is a process of converting a legacy system to a modern programming language or hardware platform. Typically, the correctness of the modernization is verified by executing a reference test suite on both the legacy and the modern system and checking the equivalence of the output values. *Regression testing* is a type of software testing that ensures that changes made to a given program do not introduces any new faults (*regressions*), *i.e.,* the program behaves correctly, as before. To conduct regression testing, the test suite for the original program is applied to newer versions of the product code to ensure that all test cases, except the ones affected by the changes, continue to pass.

To apply FSX to these use-cases, we use it to generate a full test suite, from scratch, optionally using an existing test suite as a seed, as shown in Figure 2. FSX generates test cases with test oracles that enforce (assert) the current actual output values of the program-under-test. These test cases can then be used for application modernization or regression testing purposes. In Section 6, we discuss a real industrial deployment experience of FSX for modernization of a large in-house product.

3.2 Integration with Source Code Management and Continuous Integration System

Modern Source Code Management (SCM) systems employ a peer-to-peer approach where each peer owns a complete repository and distributed repositories are synchronized by exchanging patches from peer to peer. Some systems, such as GitHub, also provide a web-based user interface, with collaboration features such as interactive patch exchange with code review, called pull requests or merge requests. Continuous Integration (CI) systems are often used in conjunction with SCM. Modern CI systems can automatically and continuously obtain the latest revision from the associated SCM system and automatically build and execute tests on the software, according to project-specific built-and-test protocols, specified in a configuration file under the source tree. Recently SCM and CI systems have become increasingly popular in industrial software development, as well as in the open-source software community.

There are several advantages of integrating FSX with SCM and CI systems. First, every time any revision is made to the product code, FSX can be autonomously invoked to revise the test code accordingly. Second, FSX can automatically identify the project configuration, from the CI configuration file, such as the code structure of product and test code and the procedures of how to build and test the software. Last, the collaboration feature of interactive patch exchange with code review is utilized for requiring users to verify test oracles in the generated test code.

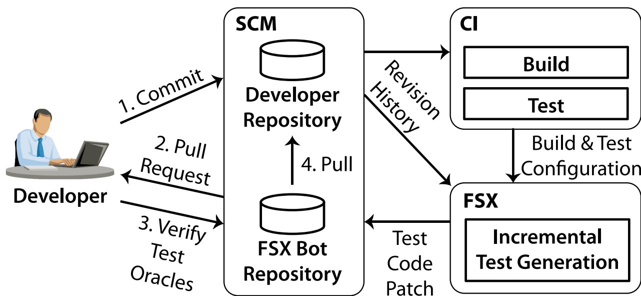

Figure 3: Integration with SCM/CI system.

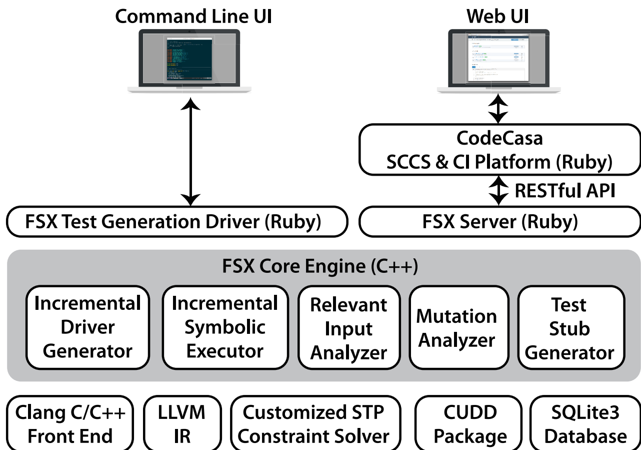

Figure 4: FSX Software Architecture.

Figure 3 illustrates how a developer interacts with FSX via an SCM system. When a developer commits a change to the product code, FSX autonomously augments the existing test suite and generates a test code patch. Shortly after the commit, the developer receives a pull request of the test code patch. In our web-based UI, the developer is requested to review the changes to the test code and to confirm or edit the expected values of the newly-introduced assertions. Once all expected values are confirmed, the developer is allowed to merge the test code in his repository. In our system, some expected values are randomly chosen and assigned erroneous values to guard against "mindless" confirmation by the developers.

4. TOOL DESCRIPTION

FSX. We implemented the techniques presented in Section 2 in our test generation tool, FSX, which has been developed from scratch with about 45,000 lines of C++ code and 7,500 lines of Ruby code. The supported platforms are Ubuntu Linux 14.04 and Mac OS X v10.9 or later. Figure 4 shows the FSX software architecture. Clang [1] is used for parsing C/C++ programs and generating its AST representation and LLVM [7] is used as the internal representation for the symbolic execution. For SMT solvers, FSX includes a customized version of STP [18] enhanced with don't care analysis [27] and also unmodified Z3 [16] which is more efficient than vanilla STP, but lacks the don't care analysis feature, used in FSX to minimize test drivers. FSXuses the CU Decision Diagram Package (CUDD) [2] for ROBDD manipulation.

FSX provides a mutation-based test oracle generation similar to the one in EvoSuite [17]. We implemented our own mutation analysis engine using the same set of mutators as PIT [9], which is a well known practical Java mutation testing tool. FSX also implements an automatic test stub generation functionality which can be used to isolate units under test and to fix missing dependencies when testing an incomplete program.

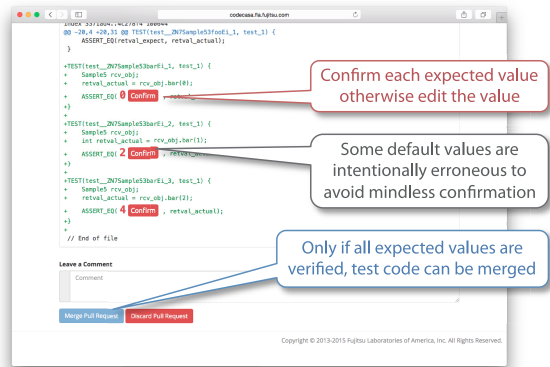

Figure 5: Web-based UI for test oracle verification.

To assure the software quality of FSX itself, it is comprehensively and frequently tested using its test-suite, developed in-house. It consists of 47 unit tests, 141 integration tests and 61 regression tests comprising 9,100 lines of test code and providing an overall line coverage of 73%.

The inputs to FSX are a program-under-test written in C or C++ and a set of previous tests written in Google Test format [4]. The outputs from FSX are two sets of tests in the Google Test format: normal tests which execute their units-under-test with normal termination and abnormal tests which are expected to terminate with abnormal termination.

Command Line UI. In Command Line mode, FSX's operation is file based and it generates unit tests for each function/method in each C/C++ source file. FSX extracts the file dependencies from the existing Makefile, allowing it to be run on any C/C++ project that builds an executable with a Makefile, by simply issuing the command:

```
fsx <fsx-options> make <make-options>
```

Web-based UI. FSX has been integrated with our in-house SCM/CI platform CodeCasa. CodeCasa is a Git-based SCM system and provides a web-based user interface. We implemented CodeCasa in Ruby and JavaScript using the Sinatra web application framework [10]. We also integrated FSX with CodeCasa according to Scenario 2 in Section 3.2. Every time a developer commits a change in a repository in CodeCasa, FSX sends a test code patch to the developer. To merge the patch, the developer needs to review the test code and to verify all test oracles, as shown in Figure 5.

5. EVALUATION

We present a representative subset of results from the full evaluation of FSX reported in [41]. The evaluation uses five revisions (v.2.0.1 - v.2.0.5) of iPerf [6], which is a widely-used open source network bandwidth measurement tool, written in C++. iPerf-2.0.x is comprised of 7 files, 39 functions, and approximately 5KLoC, *i.e.,* an aggregate of 25KLoC over the 5 versions analyzed. All experiments were run on Ubuntu 14.04 64-bit on an Intel Xeon E5-2695 v2 2.40 GHz processor with 16GB of memory. The evaluation illustrates the scenario of test-suite augmentation, explained in Section 3.2.

FSX-Baseline. We use FSX-Baseline, a simplified version of FSX, as a baseline for the experiments. It shares its symbolic execution core with FSX. The key difference is that its driver generator generates a naïve driver in a non-iterative, one-shot manner (versus the iterative, diagnosis-driven refinement in FSX), where all assignable variables including function arguments, member variables and global variables are assigned symbolic values and any pointers are set to new objects allocated in a reasonable manner.

Table 1: Test suite augmentation results from iPerf-2.0.1 to iPerf-2.0.5.

Version	FSX					FSX-Baseline				
	Statement Coverage	Branch Coverage	#Tests	Test LOC	Runtime [sec]	Statement Coverage	Branch Coverage	#Tests	Test LOC	Runtime [sec]
2.0.1	79.4%	72.1%	114	3,247	—	79.4%	72.1%	114	3,247	—
2.0.2	79.5%	72.5%	+1	+3	5.6	79.5%	72.5%	+1	+51	13.9
2.0.3	79.9%	72.9%	+5	+187	166.5	79.9%	72.9%	+5	+3,424	481.6
2.0.4	80.0%	73.0%	+2	+8	19.8	80.0%	73.0%	+2	+1,140	27.2
2.0.5	79.8%	72.9%	+3	+3	207.2	79.8%	72.9%	+3	+1,807	433.8

Table 1 shows the results of this evaluation, comparing FSX and FSX-Baseline. First, FSX is used to perform a full test suite generation (from scratch) on iPerf-2.0.1 Then each tool is used to perform test-suite augmentation using the generated test-suite of the previous version as a starting point, for each of the next 4 versions of iPerf. Thus, the generated test-suite for iPerf-2.0.1 is used as a starting point for the generation of tests for iPerf-2.0.2, and so on.

Columns $2-6$ show the test-generation results for FSX and columns $7-11$ the corresponding ones for FSX-Baseline. Columns 2 and 3 (respectively 7, 8) report the statement and branch coverage of the test-suite and column 6 (resp. 11) the total test generation time. Column 4 (resp. 9) gives the total number of *newly generated* tests (*i.e.,* added through augmentation) and column 5 (resp. 10) shows the number of lines of test code *added or modified* in the test-suite. Thus, in generating tests for iPerf-2.0.4, FSX augments the test-suite of iPerf-2.0.3 by merely adding 2 test cases and modifying 8 lines of test-code, while FSX-Baseline also adds 2 tests but needs to modify $1,140$ lines of test-code.

As shown in the table, both FSX and FSX-Baseline are able to perform test-suite augmentation and achieve comparable coverage numbers for the augmented test suite, by adding similar number of tests, per new revision. However, FSX adds or modifies significantly fewer number of lines of test code, typically $10-15$ *times* lower than FSX-Baseline, during the augmentation. This difference is because FSX, as per its design, is able to successfully re-use existing test-cases from the previous version, cloning and modifying them minimally to create new test-cases, while FSX-Baseline always generates new test-cases for uncovered functionality.

6. TOOL DEPLOYMENT

We were approached by several product divisions to improve their test development productivity, through the application of our automatic test generation tool. Most recently FSX has been deployed in an in-house application modernization project of an online vending system. This product consists of 135 Makefiles and 2,580 C/C++ source files, with a total of 504,255 executable lines of code. This product originally runs on Red Hat Enterprise Linux (RHEL) 5.5/5.6, with the target platform of this modernization project being RHEL 7.2.

This modernization project was conducted by a team according to Scenario 1 in Figure 2. Since FSX supports neither the legacy platform nor the destination modern platform, the team first ported only the modules under test to Ubuntu Linux 14.04 but not any other depending third-party modules. FSX is capable of generating test cases for such an incomplete program by automatically generating test stubs for missing dependencies. Then, the team performed a full test suite generation using FSX on Ubuntu Linux 14.04. The generated test suite achieved a statement coverage of 75.9% and branch coverage of 56.9%. The total runtime was about 14 hours. The team is currently executing the test suite on both the legacy platform and the modern platform to confirm the correctness of the modernized product.

7. RELATED WORK

Symbolic Execution based test generation. FSX builds on more then a decade of research activity on symbolic execution based test generation. Prominent representatives of this area include Symbolic PathFinder [30] for Java programs, KLEE [14] for C programs, KLOVER [25] for C++ programs, SymJS [24] for JavaScript applications, and BitBlaze [34] for security testing of program binaries. DART [19] blends concrete and symbolic execution to realize *concolic execution*, CUTE and jCUTE [33, 32] extend concolic testing to multi-threaded programs, SAGE [20] further to whole program analysis, and Pex [36] for .NET. The interested reader is referred to [29, 15] for a more complete survey. FSX's key innovation is the support for fine-grained, incremental test generation for C/C++ programs, through iterative, automatic generation of test drivers.

Incremental test generation. Several previous approaches have proposed using symbolic execution based incremental test generation for test suite augmentation. DiSE [28] and Memoise [40] focus on the characterization of changes between two program versions and running symbolic execution only on the changes. DiSE marks changes by comparing control-flow graphs of the two versions while Memoise summarizes the symbolic execution of the previous version in a *symbolic execution tree*. Xu et al. [39] use concolic execution to mutate path conditions derived from existing test-cases to exercise uncovered test targets in the new program version. eXpress [35] incorporates a similar re-use of previous tests, albeit for regression test generation.

Overall, existing SE-based incremental test generation work re-uses existing test cases *only* at the level of the path conditions the test-case embodies. FSX additionally re-uses specific test artifacts such as test drivers and test-oracles. Further, FSX's approach of iterative driver generation and incremental refinement of test-cases addresses the core problems of test-driver generation and the decision of which variables to make symbolic. These are largely not addressed in previous work.

8. CONCLUSIONS

In previous work [41], we proposed a novel technique that constructs maintainable, high-quality unit test suites by performing automated and fine-grained incremental generation of unit tests through minimal augmentation of an existing test suite. In this paper we described the tool FSX implementing that technique, including its architecture, user interface, salient features, and specific practical use-cases of this technology. We also reported on a real, large-scale deployment of FSX which we see as a practical validation of the underlying research contribution, and more broadly, of over two decades of research on automated test generation.

9. REFERENCES

[1] CLANG, http://clang.llvm.org/.

[2] CUDD: CU Decision Diagram Package, http://vlsi.colorado.edu/~fabio/CUDD/.

[3] Free Software Foundation, Inc. How to Minimize Test Cases for Bugs, https://gcc.gnu.org/bugs/minimize.html.

[4] googletest - Google C++ Testing Framework, http://code.google.com/p/googletest/.

[5] How to submit an LLVM bug report, http://llvm.org/docs/HowToSubmitABug.html.

[6] iPerf - The network bandwidth measurement tool, https://iperf.fr/.

[7] LLVM, http://www.llvm.org/.

[8] Mozilla Developer Network. Reducing testcases, https://developer.mozilla.org/en-US/docs/Mozilla/QA/Reducing_testcases.

[9] PIT Mutation Testing, http://pitest.org/.

[10] Sinatra, http://www.sinatrarb.com/.

[11] Webkit. Test Case Reduction, https://webkit.org/test-case-reduction/.

[12] B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[13] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[14] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[15] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11. ACM, 2011.

[16] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS/ETAPS'08. Springer-Verlag, 2008.

[17] G. Fraser and A. Arcuri. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[18] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07. Springer-Verlag, 2007.

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

[20] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2008.

[21] J. B. Goodenough and S. L. Gerhart. Toward a Theory of Test Data Selection. *IEEE Tran. on Software Engineering*, 1975.

[22] Y. Kim, Y. Kim, T. Kim, G. Lee, Y. Jang, and M. Kim. Automated unit testing of large industrial embedded software using concolic testing. In *28th International Conference on Automated Software Engineering (ASE)*, 2013.

[23] Y. Kim, Z. Zu, M. Kim, M. Cohen, and G. Rothermel. Hybrid directed test suite augmentation: An interleaving framework. In *2014 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2014.

[24] G. Li, E. Andreasen, and I. Ghosh. SymJS: Automatic symbolic testing of javascript web applications. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014. ACM, 2014.

[25] G. Li, I. Ghosh, and S. P. Rajan. KLOVER : A symbolic execution and automatic test generation tool for C++ programs. In *23rd International Conference on Computer Aided Verification (CAV)*, pages 609–615, 2011.

[26] P. McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2), June 2004.

[27] C. Nguyen, H. Yoshida, M. R. Prasad, I. Ghosh, and K. Sen. Generating succinct test cases using don't care analysis. In *Proc. of the 2015 IEEE 8th International Conference on Software Testing, Verification and Validation*, ICST'15, 2015.

[28] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11. ACM, 2011.

[29] C. S. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, Oct. 2009.

[30] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *25th Intl. Conf. on Automated Software Engineering (ASE)*, 2010.

[31] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *23rd International Conference on Automated Software Engineering*, ASE. IEEE, 2008.

[32] K. Sen and G. Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools. In *Proc. of the 18th International Conference on Computer Aided Verification*, CAV'06. Springer-Verlag, 2006.

[33] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *ESEC-FSE*, 2005.

[34] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *4th International Conference on Information Systems Security*, ICISS '08. Springer-Verlag, 2008.

[35] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. eXpress: Guided path exploration for efficient regression test generation. In *2011 International Symposium on Software Testing and Analysis*, ISSTA '11. ACM, 2011.

[36] N. Tillmann and J. de Halleux. PEX – White Box Test Generation for .NET. In *Tests and Proofs*, 2008.

[37] N. Tillmann and W. Schulte. Parameterized Unit Tests. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–262, 2005.

[38] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10. ACM, 2010.

[39] Z. Xu and G. Rothermel. Directed test suite augmentation. In *16th Asia-Pacific Software Engineering Conference, APSEC 2009*, 2009.

[40] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012. ACM, 2012.

[41] H. Yoshida, S. Tokumoto, M. R. Prasad, I. Ghosh, and T. Uehara. FSX: Fine-grained incremental unit test generation for C/C++ programs. In *2016 International Symposium on Software Testing and Analysis*, ISSTA 2016, 2016.