Future of Developer Testing: Building Quality in Code

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC
xie@csc.ncsu.edu

Nikolai Tillmann, Jonathan de Halleux,
Wolfram Schulte
Microsoft Research
Redmond, WA
{nikolait,jhalleux,schulte}@microsoft.com

ABSTRACT

Although much progress has been made in software verification, software testing remains by far the most widely used technique for improving software reliability. Among various types of testing, developer testing is a type of testing where developers test their code as they write it, as opposed to testing done by a separate quality assurance organization. Developer testing has been widely recognized as an important and valuable means of improving software reliability, partly due to its capabilities of exposing faults early in the development life cycle. In this position paper, we present our positions on future directions of developer testing along four dimensions (which of course we do not claim to be complete): correctness confidence, specifications, (dis)integration testing, and human factors. Our positions are originated from two recent promising technologies in developer testing: parameterized unit testing and dynamic symbolic execution, also called concolic testing.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging
General Terms: Experimentation, Human Factors, Measurement, Reliability, Theory, Verification.
Keywords: Software testing, developer testing, specifications, human factors

1. INTRODUCTION

Software reliability plays a critical role in today's businesses, governments, and society. Although much progress has been made in software verification, software testing remains by far the most widely used technique for improving software reliability. Among various types of testing, developer testing [3, 46] is a type of testing where developers test their code as they write it, as opposed to testing done by a separate quality assurance organization. Developer testing has been widely recognized as an important and valuable means of improving software reliability, partly due to its capabilities of exposing faults early in the development life cycle. The recent emerging methodology of test-driven development [10] (where unit tests are incrementally written even before the implementation code is written) further boosts the popularity of developer testing in industry. In general, developer testing helps developers to (1) gain high confidence in the code while they are writing it and (2) reduce fault-fixing cost by detecting faults early when they are freshly introduced in the code.

The popularity and benefits of developer testing have been well witnessed in industry [44]; however, manual developer testing is known to be labor intensive. In addition, manual developer testing is often insufficient in comprehensively exercising behavior of the code under test to expose its hidden faults. To address the issue, one of the common ways is to use testing tools to reduce or complement manual testing effort to achieve higher software reliability. To meet such needs, tool vendors and research labs are providing developer testing tools such as Microsoft Research Pex [40, 6] for .NET programs, Agitar AgitarOne [13, 1], Parasoft Jtest [8], and CodePro AnalytiX [2] for Java programs.

In this position paper, we present our positions on future directions of developer testing together with discussion on current research progress along four dimensions (which of course we do not claim to be complete): correctness confidence, specifications, (dis)integration testing, and human factors. We next present background information (dynamic symbolic execution and parameterized unit testing) and then discuss our positions along these four dimensions.

2. BACKGROUND

Our positions are originated from (but not limited to) two recent promising technologies in developer testing: parameterized unit testing [42, 41] and dynamic symbolic execution (DSE) [19] (also called concolic testing [36]), which combines concrete and symbolic executions to explore feasible paths in code. Note that the discussion on DSE in Sections 3-6 is also applicable on symbolic execution [27, 14] in general.

DSE is a variation of symbolic execution [27, 14] and leverages observations from concrete executions. It executes the program under test, while performing symbolic execution in parallel to collect symbolic constraints on inputs obtained from predicates in branch statements along the execution. The conjunction of all symbolic constraints along a path is called the *path condition*. DSE is typically performed iteratively to systematically increase code coverage. In each iteration, for an already explored path, DSE flips a branching node in the path to construct a new path that shares the prefix to the node with the old path, but then deviates and takes a different branch. DSE uses a constraint solver to check whether such a flipped path is feasible and if so, compute a satisfying assignment, which forms a new test input that executes along the flipped path.

In the methodology of parameterized unit testing [42, 41], developers write a parameterized unit test (PUT), which is simply a test

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

method that takes parameters, calls the code under test, and states assertions, in contrast to a conventional test method without taking any parameter. Then a test-generation tool such as Pex [40, 6] (which is based on DSE) and AgitarOne [13, 1] automatically generates test-input values for the parameters of the PUT, attempting to violate its specified assertions and achieve high code coverage of the code under test.

3. CORRECTNESS CONFIDENCE

One open question in software testing is how high confidence testers would have on program correctness after a certain amount of testing is conducted. It has been commonly believed that testing cannot provide high confidence on program correctness. A popularly quoted sentence from Dijkstra is "*Program testing can be used to show the presence of bugs, but never to show their absence!*", which is much more well known than its surrounding sentences: "*The number of different inputs, i.e. the number of different computations for which the assertions claim to hold is so fantastically high that demonstration of correctness by sampling is completely out of the question. Program testing can be used to show the presence of bugs, but never to show their absence! Therefore, proof of program correctness should depend only upon the program text.*"[1].

Developers could write a PUT, which includes assertions for asserting correctness for the program under test, which could also embed assertions. After DSE is used to generate test inputs for the PUT, high correctness confidence with respect to the assertions could likely be established. We next discuss a simplified case for illustrating this point. Let us assume that (1) the PUT (together with the invoked program under test) has a finite number of feasible paths, (2) DSE explores all these feasible paths, and (3) the constraints collected in the path condition from each iteration are within the capability of the underlying constraint solver. Then if DSE finds no violations of an assertion in the PUT, there is in fact 100% correctness confidence with respect to the assertion, i.e., the assertion is *proved* to be satisfied by *all* test inputs of the PUT.

We next give an informal sketch of the proof on the claim of 100% correctness confidence. First, we know that, for all test inputs (of the PUT) whose execution follows a path (before the assertion) explored by DSE, the assertion is proved to be satisfied; otherwise, the constraint solver would have produced test inputs whose execution follows the path and violates the assertion. Second, the whole test-input space of the PUT could be divided into disjoined partitions: the execution of the test inputs from each partition follows the same path. Finally, since DSE explores all paths and establish 100% correctness confidence for test inputs following each path, and each test input from the whole test-input space of the PUT has to follow a path, we conclude that 100% correctness confidence is established for *all* test inputs of the PUT.

Indeed, when testing real programs in practice, DSE cannot establish 100% correctness confidence for two main reasons. First, DSE may not be able to explore a large number of all feasible paths within reasonable time (in fact, there could be an infinite number of feasible paths for exploration, e.g., in the presence of a loop whose iteration bound is an unbounded integer input). Second, some constraints from a path condition could be too complex for the underlying constraint solver to solve.

Below are example open research questions related to correctness confidence (especially test generation):

- How do we improve DSE to deal with those barriers in compromising correctness confidence? One significant challenge is the path explosion problem. Recent techniques are proposed to alleviate issues related to loops [35, 48], string constraint solving [26, 43], and object creation [39].
- How do we measure and report to the developers the level of correctness confidence after DSE is applied (when it is not 100%, which could be often the time in practice)? For a measured and reported level, how do we validate that it reflects the real level? On possible direction is to adapt or customize traditional code coverage [49] to take into account of the assertion under consideration and the difficulties faced by DSE. For example, the coverage of some not-yet-covered branches may have no impact on the (un)satisfaction of the assertion under consideration, and then these branches could be excluded for measurement. One example direction is to focus the measurement of code coverage on only the problematic areas for DSE to reflect the compromised confidence levels.
- How do we improve the level of correctness confidence after DSE is applied and 100% correctness confidence cannot be established? One example direction is to use static verification to complement testing [20, 11].

In addition, keep in mind that whatever level of correctness confidence being established is with respect to the assertion under consideration. There could be a gap between the confidence on the assertions (written in the PUT or embedded in the program under test) and the confidence on the program correctness. Such issues are discussed in the next section on specifications.

4. SPECIFICATIONS

White-box testing has been known to be ineffective in detecting omission faults, which are related to missing functionalities or logics. One example is that if the program under test misses a certain branch, e.g., an input-validation branch for checking whether an input x is greater than 100, then there exists no such a path involving this missing branch in the program under test. Then DSE does not explore such a path involving this missing branch, and the constraint solver may not specifically generate test inputs where x is greater than 100. On the surface, this observation seems to contradict to our discussion in the preceding section on the claim of 100% correctness confidence with DSE, which is basically a white-box testing technique.

What is the catch here? When DSE is applied, assertions in the PUT or the program under test change the picture: if we write an assertion to assert the return value of the program under test to be greater than 0 (assuming that missing that input-validation branch could cause this assertion to be violated for test inputs greater than 100), applying DSE still allows us to detect the omission fault. However, if *no* assertions are written, and developers rely on inspecting whether the actual output (e.g., return value) from the execution of each single generated test input (one for each explored path when applying DSE) is expected, we cannot escape the fate of not being able to detect omission faults.

A PUT or the program under test with assertions in combination of a white-box test-generation tool (such as one based on DSE) can be classified as an integrated form of both white-box testing and black-box testing. The assertions make the conducted testing to enjoy the benefits of black-box testing, alleviating the well-known limitation of white-box testing to deal with omission faults.

Assertions in a PUT or the program under test can be seen as a form of specifications. In the context of developer testing, two common types of specifications are algebraic specifications [21] and axiomatic specifications [23] (also called design by contracts [33]).

[1] `http://userweb.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD268.html`

Axiomatic specifications for object-oriented programs are in the form of method preconditions, method postconditions, and class invariants. Jtest, AnalytiX, and Pex (when used in combination of Code Contracts [5]) support such types of axiomatic specifications. AgitarOne allows developers to specify class invariants. Algebraic specifications can be encoded in PUTs supported by Pex and AgitarOne. Note that developers could encode specifications in the form of assertions within PUTs, beyond traditionally defined algebraic specifications. In the rest of this position paper, we refer to specifications that could be encoded in PUTs as algebraic specifications (defined in a loose sense).

We next summarize two main differences between axiomatic and algebraic specifications in providing test oracles for developer testing. First, axiomatic specifications are written and placed in the code under test and the specifications could refer to internal implementation information (e.g., private fields), whereas algebraic specifications are written and placed in the test code (e.g., PUTs) and the specifications could refer to only information visible outside of the interface of the code under test, e.g., often referring to two public methods in code interface (e.g., `push` and `pop` methods of a `Stack` class). Second, in practice, axiomatic specifications are often written to capture general behavior of the code under test, whereas algebraic specifications in PUTs are written to capture behaviors applicable to scenarios (encoded in the PUTs) where the code under test is exercised (e.g., asserting the behavior of `pop` invoked immediately after `push` is invoked). It is believed that writing (general) axiomatic specifications is more difficult than writing (scenario-specific) algebraic specifications.

Below are example open research questions related to specifications:

- What are costs and benefits in writing axiomatic versus algebraic specifications? What are the limitations of axiomatic versus algebraic specifications in expressing behavior of code under test in practice? Are there any other specification types that could be used to complement axiomatic and algebraic specifications in serving as test oracles? For what types of code under test, developers are advised to write axiomatic versus algebraic specifications? For example, axiomatic specifications may be written better for reusable library or framework code so that one-time substantial investment of writing specifications could get paid off with many library or framework API clients being checked with the specifications.
- Could inferred specifications [17, 15] be good enough for serving as test oracles when specifications are not manually written? AgitarOne recommends specification candidates for developers to confirm. However, these specification candidates are often limited, e.g., being often insufficient to capture real method postconditions, which could be quite complex. Indeed, these inferred specification candidates could serve as stimulus for encouraging developers to write down more specifications, to complement the recommended specification candidates.
- How could we measure the quality of written specifications for the code under test? High fault-detection capability cannot be accomplished even when perfect test-generation tools are available but written specifications are insufficient. There existed initial work to attempt to address the question with mutation analysis in the area of verifying security policies [32] and with analysis of object-field read/write [37].

5. (DIS)INTEGRATION TESTING

In developer testing, unit testing, i.e., testing individual components in isolation, is primarily conducted. In real-world code bases, a component could have quite some dependencies on external environments such as file systems and complex frameworks for dealing with webs or clouds. A common solution is to mock or simulate environment dependencies [30, 28, 31, 38], so that the unit tests run quickly and give deterministic results. Basically, such a solution improves the testability [18], including controllability and observability: developers could better control what environment-input values a unit test needs to have, and developers could add assertions at the environment-interaction points for checking whether values flowing to the environment are expected. We name such testing as *disintegration testing* (since it aims to break the integration, being contrary to integration testing). There existed tool support [16] for isolating the environment. Developers could also spend one-time effort for writing parameterized models [28, 31, 38] for a specific environment to faithfully simulate the behavior of the environment. Then when testing any client code interacting with that environment, the same developer or other developers could reuse such parameterized models.

In developer testing, integration testing would still need to be conducted, whose main goal is to test the integration of different components, including environments that components interact with. Integration testing should be conducted after unit testing is conducted.

Below are example open research questions related to (dis)integration testing:

- How could we measure the quality of parameterized models for faithfully simulating an environment? Is there any methodology for systematically modeling the environment with parameterized models? Parameterized models provide an abstraction of the environment, capturing only essential behaviors from the perspective of components interacting with the environment. Modeling techniques in model-based testing [24] could be borrowed to tackle these questions.
- How could we make smooth transition from disintegration testing (i.e., isolated unit testing) to integration testing by exploiting the knowledge gained in unit testing of isolated components? One possible direction [38] is to automatically synthesize a real-environment state for the simulated-environment state after it is generated during disintegration testing; then the real-environment state could be used during integration testing.
- What types of components besides environments should developers mock? In principle, developers could mock any component (not just environments) that the component under test interacts with. Potentially when a dependent component is too complex to explore with DSE, developers could mock this component. However, mocking comes with cost: it takes effort to write faithful parameterized models to prevent infeasible paths (which could cause false warnings among test failures).

6. HUMAN FACTORS

Tool automation such as improving automated test generation [46] has been a traditional focus of software testing research. At the same time, human factors also play important roles in software testing. For example, given the code under test, tools can try to automate the generation of test inputs as much as possible but test oracles still need come from developers or testers, who specify them in the form of specifications (or generally assertions), or directly inspect the actual test outputs for correctness. In addition, tools are not always perfect to deal with software complexity; developers or testers need to cooperate with tools to effectively carry out testing

tasks, by interpreting results produced by tools and giving guidance to the tools. In addition, to effectively carry out these tasks, developers or testers need to be well educated or trained to possess necessary skills. Taking human factors together with tool automation, we advocate a new methodology of *cooperative developer testing*[2], where tools and developers cooperate to effectively carry out developer testing. We recently co-organized a 2010 Dagstuhl Seminar [22] on "Practical Software Testing: Tool Automation and Human Factors"[3].

Below are example open research questions related to human factors:

- How could tools be designed to effectively explain the challenges that the tools face and seek guidance from developers (if at all possible)? One possible direction [45] is to provide accurate explanation and narrow down the investigation scope for developers (thus reducing their required effort). Effective visualization [25, 29, 12] of testing information such as coverage is also an important direction.
- Where and how could we provide effective ways for developers to guide tools? Some example guidance to tools given by developers include writing factory methods [40, 6, 13, 1, 8, 2] that include method sequences for generating desirable objects, a common challenge faced by existing tools [39], as well as instructing tools on what classes to instrument for path exploration and mocking specific APIs [16].
- How could we provide effective tool support to help developers write and debug high-quality specifications? How could we design effective teaching/training methods [47] as well as educational materials and tools [7] for teaching/training developers to write high-quality specifications? With the adoption of DSE and PUTs, we believe that the traditional testing curriculum with already good focus on coverage criteria [9] and black-box testing methodologies should be extended to put more emphasis on specification writing and abstract thinking.

7. CONCLUSION

In this position paper, we have laid out our positions in future of developer testing from the dimensions of correctness confidence, specifications, (dis)integration testing, and human factors. Our positions are originated from two recent promising technologies in developer testing: parameterized unit testing and dynamic symbolic execution, also called concolic testing.

Our discussion has been primarily on functional correctness. However, all or most of our discussed positions could be also applicable for other quality attributes such as security and performance. Indeed, these other quality attributes would call for further new future directions. Our discussion has been primarily on testing sequential code. Concurrency testing [34, 4] is also important in developer testing, and would also call for further new research directions.

Acknowledgments

Tao Xie's work is supported in part by NSF grants CNS-0716579, CCF-0725190, CCF-0845272, CCF-0915400, CNS-0958235, an NCSU CACC grant, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU SOSI.

[2] https://sites.google.com/site/asergrp/projects/cooptest

[3] http://www.dagstuhl.de/programm/kalender/semhp/?semnr=10111

8. REFERENCES

[1] Agitar AgitarOne. http://www.agitar.com/.

[2] CodePro AnalytiX. http://old.instantiations.com/codepro/analytix/default.htm.

[3] Developer testing. http://www.developertesting.com/.

[4] Microsoft Research. CHESS. http://research.microsoft.com/projects/chess/.

[5] Microsoft Research. Code Contracts. http://research.microsoft.com/projects/contracts.

[6] Microsoft Research. Pex. http://research.microsoft.com/projects/pex/.

[7] Microsoft Research. Pex for Fun. http://www.pexforfun.com/.

[8] Parasoft Jtest. http://www.parasoft.com/jsp/products/home.jsp?product=Jtest.

[9] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[10] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.

[11] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *Proc. International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 3–14, 2008.

[12] S. Berner, R. Weber, and R. K. Keller. Enhancing software testing by judicious use of code coverage information. In *Proc. International Conference on Software Engineering (ICSE 2007)*, pages 612–620, 2007.

[13] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *Proc. International Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 169–180, 2006.

[14] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[15] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: dynamic symbolic execution for invariant inference. In *Proc. International Conference on Software Engineering (ICSE 2008)*, pages 281–290, 2008.

[16] J. de Halleux and N. Tillmann. Moles: tool-assisted environment isolation with closures. In *Proc. International Conference on Objects, Models, Components, Patterns (TOOLS 2010)*, pages 253–270, 2010.

[17] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transaction on Software Engineering*, 27(2):99–123, 2001.

[18] R. S. Freedman. Testability of software components. *IEEE Transaction on Software Engineering*, 17(6):553–564, 1991.

[19] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 213–223, 2005.

[20] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: a new algorithm for property checking. In *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2006)*, pages 117–127, 2006.

[21] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[22] M. Harman, H. Muccini, W. Schulte, and T. Xie. 10111 executive summary – practical software testing: Tool automation and human factors. In M. Harman, H. Muccini, W. Schulte, and T. Xie, editors, *Practical Software Testing : Tool Automation and Human Factors*, number 10111 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[23] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[24] J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-Based Software Testing and Analysis with C#*. Cambridge University Press, 2007.

[25] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. International Conference on Software Engineering (ICSE 2002)*, pages 467–477, 2002.

[26] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In *Proc. International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 105–116, 2009.

[27] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[28] S. Kong, N. Tillmann, and J. de Halleux. Automated testing of environment-dependent programs - a case study of modeling the file system for Pex. In *Proc. International Conference on Information Technology: New Generations (ITNG 2009)*, pages 758–762, 2009.

[29] J. Lawrance, S. Clarke, M. Burnett, and G. Rothermel. How well do professional developers test with code coverage visualizations? an empirical study. In *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC 2005)*, pages 53–60, 2005.

[30] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: Unit testing with mock objects. In *Extreme Programming Examined*. Addison-Wesley, 2001.

[31] M. R. Marri, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. An empirical study of testing file-system-dependent software with mock objects. In *Proc. International Workshop on Automation of Software Test (AST 2009), Business and Industry Case Studies*, pages 149–153, 2009.

[32] E. Martin, J. Hwang, T. Xie, and V. Hu. Assessing quality of policy properties in verification of access control policies. In *Proc. Annual Computer Security Applications Conference (ACSAC 2008)*, pages 163–172, 2008.

[33] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[34] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, pages 267–280, 2008.

[35] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proc. International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 225–236, 2009.

[36] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. the 5th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, pages 263–272, 2005.

[37] Y. Song, S. Thummalapenta, and T. Xie. UnitPlus: Assisting developer testing in Eclipse. In *Proc. the Eclipse Technology eXchange Workshop at OOPSLA 2007 (ETX 2007)*, pages 26–30, 2007.

[38] K. Taneja, Y. Zhang, and T. Xie. MODA: Automated test generation for database applications via mock objects. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, short paper, 2010.

[39] S. Thummalapenta, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proc. the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009)*, pages 193–202, 2009.

[40] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *Proc. International Conference on Tests & Proofs (TAP 2008)*, pages 134–153, 2008.

[41] N. Tillmann, P. de Halleux, and T. Xie. Parameterized unit testing: Theory and practice. In *Proc. International Conference on Software Engineering (ICSE 2010), Companion Volume, Tutorial*, pages 483–484, 2010.

[42] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proc. the 5th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, pages 253–262, 2005.

[43] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Proc. IEEE International Conference on Software Testing, Verification and Validation (ICST 2010)*, pages 498–507, 2010.

[44] G. Venolia, R. DeLine, and T. LaToza. Software development at microsoft observed. Technical Report MSR-TR-2005-140, Microsoft Research, Redmond, WA, October 2005.

[45] X. Xiao, T. Xie, N. Tillmann, and P. de Halleux. Issue analysis for residual structural coverage in dynamic symbolic execution. Technical Report TR-2010-7, North Carolina State University Department of Computer Science, Raleigh, NC, March 2010.

[46] T. Xie. Improving automation in developer testing: State of the practice. Technical Report TR-2009-6, North Carolina State University Department of Computer Science, Raleigh, NC, February 2009.

[47] T. Xie, J. de Halleux, N. Tillmann, and W. Schulte. Teaching and training developer-testing techniques and tool support. In *Proc. Annual ACM Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH 2010), Educators' and Trainers' Symposium*, 2010.

[48] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, pages 359–368, June-July 2009.

[49] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.