Improving Automation in Developer Testing: State of the Practice

Tao Xie
Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
xie@csc.ncsu.edu

Abstract

Developer testing, a common step in software development, involves generating desirable test inputs and checking the behavior of the program unit under test during the execution of the test inputs. Existing industrial developer testing tools include various techniques to address challenges of generating desirable test inputs and checking the behavior of the program unit under test. This paper presents an overview of techniques implemented in industrial developer testing tools to address challenges in improving automation in developer testing. These techniques are summarized from two main aspects: test efficiency (e.g., with a focus on cost) and test effectiveness (e.g., with a focus on benefit).

1 Introduction

Software testing is the most widely used approach for improving software quality in practice. Among various types of testing, developer testing (where developers test their code as they write it, as opposed to testing done by a separate quality assurance organization) has been widely recognized as a valuable means of improving software quality. Developer testing, often in the form of unit testing, helps developers to (1) gain high confidence in the the program unit under test (e.g., a class) while they are writing it and (2) reduce fault-fixing cost by detecting faults early when they are freshly introduced in the program unit.

The popularity and benefits of developer testing have been well witnessed in industry; however, manual developer testing is known to be labor intensive. In addition, manual testing is often insufficient in comprehensively exercising behavior of the program unit under test to expose its hidden faults. For example, due to program complexity and limited human-brain power, developers may not be capable of coming up with certain test inputs (such as corner or special test inputs) that can expose faults in the program unit.

To address the issue, one of the common ways is to conduct automated developer testing by using tools to automate activities in developer testing. Developer testing activities typically include generating test inputs, creating expected outputs, running test inputs, and verifying actual outputs. Developers can use existing testing frameworks such as JUnit [9] to write unit-test inputs and their expected outputs. Then JUnit can automate running test inputs and verifying actual outputs against the expected outputs.

To reduce the burden of manually creating test inputs, developers can use industrial testing tools to generate test inputs automatically. But then expected outputs for these test inputs are still missing, and it is infeasible for developers to create expected outputs for this large number of generated test inputs. Specifications [12] can be used to improve the effectiveness of generating test inputs and check program behaviors when running test inputs without expected outputs. Without requiring specifications (which may be difficult for developers to write), testing tools can use code coverage criteria such as statement coverage and block coverage to select a subset of generated test inputs for developers to manually verify the actual outputs.

This paper presents an overview of state of practice in improving automation in developer testing. We provide insights to major industrial tools' key features in improving automation in developer testing. We collect a list of major industrial developer testing tools from various sources. Specifically, we collect the first list of developer testing tools among testing tool finalists of recent annual Jolt Product Excellence and Productivity Awards[1], prestigious industrial awards in recognizing excellent industrial products. We also include some other industrial tools that first adopted important features later incorporated by some tools in the first list. In the end, this paper discusses (1) three industrial tools among Jolt Award finalists: Parasoft Jtest [14] for Java, Agitar AgitarOne [4, 5] for Java, CodePro AnalytiX [1] for Java, and (2) two other industrial tools: Microsoft Pex [3] for C# and SilverMark Test Mentor [18] for Java. Throughout the discussion of features provided by these tools, we also briefly mention selected relevant research work from academia that may help fill the gap left by existing industrial tools. Note that the information for these discussed industrial tools is drawn from the public

[1] http://www.joltawards.com/

domain (e.g., from tool materials in respective vendor web sites). This paper does not intend to compare these industrial tools side by side or provide ranking among these tools, but highlight valuable features provided by these industrial tools from two main aspects: test efficiency (e.g., with a focus on cost) and test effectiveness (e.g., with a focus on benefit).

2 Test Efficiency

Existing tool support for improving test efficiency includes creating and running test inputs more efficiently. One main technique in improving test efficiency is the *capture-and-replay* technique [6, 13, 16]. Such a technique has been traditionally used in GUI or Web application testing, being supported by various industrial tools such as IBM Rational Robot [15].

In the context of developer testing, the *capture* phase of the technique monitors the interactions of the unit under test (e.g., a class) and its environment (e.g., the rest of the system where the class is) during the execution of the system. Such system execution can be induced by manually or automatically running system tests. Based on the monitored interactions, the capture phase automatically creates unit tests for the unit under test. Each unit test includes (1) test inputs as captured method invocations of the unit (in addition to some other necessary method invocations of other units) and (2) test oracles as the captured return values of the method invocations of the unit. The *replay* phase of the technique simply reruns the created unit tests, which check the unit behavior with their test oracles.

In contrast to automatically running the system tests, automatically running the created unit tests is faster since the unit tests focus on only the interactions with the unit under test. In contrast to manually running the system tests, automatically running the created unit tests is much faster since no manual effort is required besides focusing on the interactions with the unit. In contrast to manually writing these unit tests for the unit, the technique allows automatic creation of these unit tests.

The technique would be primarily useful in regression testing (i.e., checking the behavior of a new version to be the same as the one of the old version). When applying the technique on an initially faulty unit, the capture phase would capture the *faulty* behavior of the faulty unit and the replay phase would simply make sure that this faulty behavior would remain in future versions!

Note that this technique exercises no new unit behavior beyond the one exercised by the system-test execution. Therefore, no new code coverage can be achieved by the created unit tests beyond the system tests. But it may be possible that the created unit tests can expose new faults not exposed by the system tests since the test oracles in the unit tests can be stronger than the ones for the system tests by checking inside the system black box [24].

Some industrial tools adopt the capture-and-replay technique. For example, SilverMark Test Mentor provides a feature called *Object Interaction Recording*. Parasoft Jtest provides a feature called *Jtest Tracer* (previously called *Test Case Sniffer*).

One challenge in this technique is to deal with non-primitive argument values and return values on the unit interface, when creating unit tests from captured unit interactions. Some tools may just handle only primitive values in the unit interactions. Some tools may serialize an object's value in the capture phase and deserialize it in the replay phase. But such a mechanism would produce obsolete or broken unit tests for later versions of the unit, e.g., when classes related to these objects are refactored, causing their object fields being changed. One better mechanism is to capture and replay method sequences that produce actual object values [23]. Another better mechanism is to use a mock object [16] in place of an argument or return object (as supported by Microsoft Pex [3, 20] and related to automatic stub generation provided by Parasoft Jtest [14]). Then tools can capture and replay arguments and return values of methods of the mock object. Various researchers [6, 13, 16] have investigated advanced mechanisms of the capture-and-replay technique. In terms of improving efficiency in manually creating unit tests, researchers [17] have also developed IDE support for helping developers write unit test inputs and oracles faster.

3 Test Effectiveness

In contrast to improving test efficiency (concerning more about reducing cost), improving test effectiveness focuses on improving the quality of test inputs and test oracles. We next discuss techniques and challenges in generating test inputs and test oracles.

3.1 Test Inputs

In unit testing of object-oriented programs, test-input generation includes two subtasks: (1) generating desirable primitive-method-argument values and (2) generating desirable method-argument objects and receiver objects. Here *desirable* test inputs are test inputs for achieving some test objectives not previously achieved such as causing program crashes, violating specified properties, and achieving high code coverage (which are all related to test oracles discussed in Section 3.2).

In the first subtask, industrial tools use four main techniques: using predefined values for a primitive type, using constant values (of the same primitive type) that appear in the code under test, using random values, and using values derived with symbolic execution [11]. Symbolic execution has been popularly used by tools such as Parasoft Jtest, Agitar AgitarOne, and Microsoft Pex. The symbolic execution technique generally performs better than the other three main techniques in terms of achieving high code coverage.

In the second subtask, industrial tools use three main techniques on generating method sequences to produce desirable objects: generating default sequences (e.g., using the null reference and invoking only a constructor), generating random method sequences, generating (quasi)bounded-exhaustive sequences, and generating sequences based on heuristics. Most industrial tools may use one or more of the first three techniques where the third technique often uses only a small bound (e.g., up to method sequences of length 3). Microsoft Pex uses heuristics based on collected code information to generate desirable method sequences.

In test-input generation at the unit-testing level, one issue is to deal with illegal test inputs (also called invalid test inputs), which are test inputs that the unit under test is not expected to handle. Adopting the design-by-contract methodology [12], Parasoft Jtest, CodePro AnalytiX, and Microsoft Pex (when used in combination of Microsoft Contracts [2]) allow developers to specify method preconditions or class invariants for the unit under test. Then their test generation engines would filter out or avoid generating illegal test inputs, which violate method preconditions or class invariants. Microsoft Pex also allows developers to write assumptions for parameterized unit tests (PUTs) [19] (unit test methods with parameters, more details described in Section 3.2.2). Then assumption-violating test inputs for PUTs are filtered out or avoided in test-input generation.

Some tools such as Agitar AgitarOne adopt a defensive programming methodology, where developers are advised to write explicit checking code in the beginning of a method body of the unit under test to detect illegal test inputs, (once detected) throwing appropriate exceptions. Note that these tools' test-input generation engines would still generate illegal test inputs.

Challenges in the first subtask include complex logics in code involving complicated constraints beyond the capability of constraint solvers underlying symbolic execution, and involving a large/infinite number of paths (especially when loops are involved). Challenges in the second subtask are even more substantial since it faces even more daunting combinations of possible method calls (each of which may need the first subtask to resolve its primitive argument values). Researchers [10] have explored advanced techniques to address challenges in method-sequence generation.

One direction to address challenges faced in test-input generation is to allow developers to guide tools in different ways. For example, developers can specify data factories for a non-primitive object type. Such data factories are called test-input factories or test helpers in Agitar AgitarOne, object repositories in Parasoft Jtest, and factory classes or methods in CodePro AnalytiX and Microsoft Pex. As another example, CodePro AnalytiX and Parasoft Jtest allow developers to directly edit the generated test inputs to improve them. Microsoft Pex allows developers to write parameterized unit tests [19] (e.g., where developers can write desirable method sequences with primitive values being unspecified and generated by Pex). Researchers [21, 25] have also explored techniques for exploiting information from manually written traditional unit tests in guiding the generation of new test inputs.

3.2 Test Oracles

There are two main levels of test oracles: ones specific only to one individual test input and ones applicable to multiple test inputs.

3.2.1 Test Oracles Specific to Individual Test Input

Test oracles can be in the form of assertions in manually written unit tests (such as those in JUnit [9]). Developers can relatively easily write assertions for one test input but writing them for many unit-test inputs (which can be generated by tools) is time-consuming and infeasible.

In the regression testing context, similar to the capture-and-replay technique described in Section 2, tools can use the capture-and-assert technique [22], which captures the return values of methods of the unit under test during the execution of the generated test inputs, and then automatically creates assertions based on the captured return values. Parasoft Jtest, CodePro AnalytiX, Agitar AgitarOne, and Microsoft Pex implement such technique for regression testing. A resulting test case with automatically created assertions is called a characterization test by Feathers [8].

In general testing (beyond regression testing), tools such as Parasoft Jtest, CodePro AnalytiX, and Microsoft Pex allow developers to inspect and verify captured assertions in the generated unit test suite. To reduce inspection effort, these tools allow to select only test inputs that can achieve new code coverage (such as statement coverage and block coverage) not previously achieved.

3.2.2 Test Oracles Applicable to Multiple Test Inputs

Test oracles applicable to multiple test inputs can be classified into three types. First, developers can determine whether the execution of a test input fails based on whether the execution throws an uncaught exception. Such type of test oracles is related to robustness testing, being supported by all industrial tools. Note that this type of test oracles is quite limited since the execution may throw no uncaught exceptions but produce wrong outputs (such as wrong method-return values).

Second, developers can write *properties* (also called specifications) in *the unit code under test* based on the design-by-contract methodology [12], where properties can be in the form of method preconditions, method postconditions, and class invariants. Parasoft Jtest, CodePro AnalytiX, and Microsoft Pex (when used in combination of Microsoft Contracts [2]) support such type of test oracles. Agitar AgitarOne allows developers to specify class invariants. Developers may have a more difficult time in writing a property than writing an assertion in unit tests (as in the first level of test oracles) but one property can be used to check the execution of multiple test inputs, offering more benefits.

To reduce the difficulty of writing properties and stimulate developers to write properties, Agitar AgitarOne implements a software agitation technique [5] based on dynamic invariant detection [7]. It generates observations of the code behaviors inferred from the execution of automatically generated test inputs. These observations summarize common behavioral patterns reflected by the execution of multiple test inputs. Then developers can inspect and verify these observations: if these observations reflect desirable behaviors, developers promote them to be assertions (such as method postconditions and class invariants); if these observations reflect faulty behaviors, developers detect faults and then fix these faults.

Third, developers can write *properties* in *unit test code* such as parameterized unit tests (PUTs) [19], which are test methods with parameters. Developers can write assumptions (similar to preconditions) and assertions (similar to postconditions) in PUTs; however, assumptions or assertions in PUTs are often not specified for only one specific method but for a scenario where multiple methods are invoked. Both Microsoft Pex and Agitar AgitarOne support parameterized unit tests. To some extent, writing assertions in PUTs can be viewed as in a middle ground between writing assertions in traditional unit tests and writing properties in the unit code under test, in terms of test oracles' fault-detection capability, ease of writing, and scope of benefits.

References

[1] CodePro AnalytiX, Accessed in January 2009. `http://www.instantiations.com/codepro/`.

[2] Microsoft Research Contracts, accessed in January 2009. `http://research.microsoft.com/en-us/projects/contracts`.

[3] Microsoft Research Pex, accessed in January 2009. `http://research.microsoft.com/en-us/projects/Pex/`.

[4] Agitar AgitarOne, Accessed in January 2009. `http://www.agitar.com/solutions/products/agitarone.html`.

[5] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *Proc. ISSTA*, pages 169–180, 2006.

[6] S. Elbaum, H. N. Chin, M. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *Proc. FSE*, pages 253–264, 2006.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

[8] M. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004.

[9] E. Gamma and K. Beck. JUnit. `http://www.junit.org`.

[10] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proc. ASE*, pages 297–306, 2008.

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

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

[13] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Proc. WODA*, pages 29–35, 2005.

[14] Parasoft Jtest, accessed in January 2009. `http://www.parasoft.com/jsp/products/home.jsp?product=Jtest`.

[15] IBM Rational Robot, accessed in January 2009. `http://www-01.ibm.com/software/awdtools/tester/robot/index.html`.

[16] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proc. ASE*, pages 114–123, 2005.

[17] Y. Song, S. Thummalapenta, and T. Xie. UnitPlus: Assisting developer testing in Eclipse. In *Proc. ETX*, pages 26–30, 2007.

[18] SilverMark Test Mentor, accessed in January 2009. `http://www.silvermark.com/Product/java/stm/`.

[19] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proc. ESEC/FSE*, pages 253–262, 2005.

[20] N. Tillmann and W. Schulte. Mock-object generation with behavior. In *Proc. ASE*, pages 365–368, 2006.

[21] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Softw.*, 23(4):38–47, 2006.

[22] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. ECOOP*, pages 380–403, 2006.

[23] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. ASE*, pages 196–205, 2004.

[24] T. Xie and D. Notkin. Checking inside the black box: Regression testing by comparing value spectra. *IEEE Trans. Softw. Eng.*, 31(10):869–883, October 2005.

[25] T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 13(3):345–371, July 2006.