From Developer's Head to Developer Tests

Characterization, Theories, and Preventing One More Bug

David Saff
MIT CSAIL
saff@mit.edu

Abstract

Unit testing frameworks like JUnit are a popular and effective way to prevent developer bugs. We are investigating two ways of building on these frameworks to prevent more bugs with less effort. First, *characterization* tools summarize observations over a large number of executions, which can be checked by developers, and added to the test suite if they specify intended behavior. Second, *theories* are developer-written statements of correct behavior over a large set of inputs, which can be automatically verified. We outline an integrated toolset for characterization and theory-based testing, and frame further research into their usefulness.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—tools

General Terms Verification, Human Factors

Keywords Theories, JUnit, testing, partial specification

1. Introduction

Every software feature is intended to match an "internal specification" in the head of its developer. How can the developer ensure that the initial version, and all future revisions, hold to this specification over all valid inputs? Manually executing the feature and examining the software's output and behavior can find *mispredictions*, in which the developer correctly anticipates valid input, but fails to correctly predict how the written software will behave.

To prevent mispredictions in future versions, the developer can write an automated test, including the sample inputs and expected behavior, using a testing framework like JUnit[1], which has a simple interface and builds on a developer's familiarity with the language of the domain code. This discipline of developer testing can be effective for a wide range of skill levels. However, bugs may still linger, through *oversights*: valid inputs that the developer has failed to properly anticipate. Once an incorrect output for a valid input is brought to the developer's attention, it is easy to write a test for the desired behavior, and debug as before. [2] How confident can the developer be that enough possible inputs have been considered, that there are not more bugs lurking in the unexplored regions of the input space? Manual techniques based on equivalence classes and code coverage can be effective, but time consuming, and may need to be repeated after revisions.

We are evaluating two ways to automatically support the search for oversight bugs. *Characterization* executes a feature many times, and selects individual executions or summary properties of the code to bring to the developer's attention. *Theory exploration* takes as input general statements of correctness, written as extensions of traditional unit tests, and searches the input space for violations.

2. Characterization

A characterization tool initiates or observes many different executions of a sofware feature and presents a summary of the feature's behavior. The developer is responsible for recognizing which behaviors in the summary are desirable (and can therefore be enshrined as regression tests), and which indicate bugs.

A computer can only search a fraction of a potentially infinite input space, and a human can only devote attention to a much smaller fraction. To use these resources wisely, a characterization tool must cleverly choose inputs that are likely to yield interesting insights into the code's behavior. The characterization tools we are investigating use the input generation facilities of Agitator [1]. Agitator uses a pragmatic combination of established techniques, including symbolic execution, constraint solving, heuristics, randomization, and gleaning constructed objects from the subject code.

Once thousands of appropriate inputs are chosen, executed, and the outputs recorded, what should be presented to the developer? Daikon [2] and Agitator [1] present sum-

[1] http://junit.sourceforge.net

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-786-5/07/0010.

[2] We do not consider here bugs created through *miscommunication*, in which the developer's internal specification is actually wrong when compared to the user's needs.

maries in the form of *invariants*: statements of program behavior that have held true over every observed execution, such as `x[0] != null`. A bug may manifest itself through an invariant that doesn't match the developer's internal specification, or through an invariant that should have been detected, but is not.

Alternatively, Agitar's free JUnit Factory[3] service finds a minimum set of input sequences necessary to maximize code coverage, and presents each input sequence as a JUnit test. The potential advantage of this approach is that specific examples may be easier to examine for correctness than invariants, and the output language is one the developer is already familiar with.

3. Theory-based testing

In characterization, the developer supplies no up-front correctness criteria, but examines the output for potential problems. By turning this process around, it may be possible to better use the developer's time. The developer hypothesizes once and for all a *theory*, a desired general property of program execution. The developer can verify the theory on a few hand-picked inputs, but automated tools can *explore* the input space after every code revision, experimenting with possible values to find any violations

Automatically verifying specifications is not new. Model-checking tools can search for violations of specifications expressed using specialized property objects or logic languages. Agitator can also search for violations of important invariants.

However, we believe that to spread the benefits of theory exploration to more developers, the interface must be very simple, requiring little additional expertise beyond unit testing. Therefore, we have built on Tillman and Schulte's work on Parameterized Unit Tests [4] to provide Theories, a new testing construct included in JUnit as of version 4.4.

The built-in Theory support in JUnit can verify theories against a list of data points provided by the developer. A forthcoming open-source tool, Theory Explorer, can use an input generator like Agitar's to find new data points that violate the theory, and add them to the standard list. For more about Theories, their uses, and relationship to previous work, please see our companion paper. [3]

4. Tools

For developers and researchers to compare the usefulness of characterization and theories to traditional testing discipline, it is useful to have a set of tools that speak the same visual and logical language. Therefore, we are developing and collecting a suite of Java tools that build on the standard JUnit framework to provide these new services. Two of these tools already exist:

- JUnit 4.4 provides automatic execution of traditional tests, as well as Theories, when provided with interesting data points by the developer.
- JUnit Factory provides characterization of software behavior as generated JUnit tests. JUnit Factory is free for software without intellectual property restrictions, or available as part of the commercial AgitarOne product.

Two are currently in development:

- Theory Factory provides characterization as generated JUnit Theories, which are based on the invariants discovered by Agitator.
- Theory Explorer explores the input space of JUnit Theories using an input-generation engine such as Agitator.

5. Questions

With case studies using these tools, we plan to investigate several questions:

1. What kinds of bugs are more likely to be found through code inspection and manual testing discipline than through automatic characterization, and vice versa?
2. What is the time investment for manually creating a test compared to properly verifying a machine-generated test?
3. Are developers more likely to recognize a bug exposed by characterization if the characterization results are expressed using tests or theories?
4. When is it easier to think through a new requirement with a single example test than with a general statement theory, and vice-versa?
5. How useful are theories for a developer generating new testing ideas, even before using automated exploration?

6. Acknowledgement

This work is supported by an Agitar Research Fellowship.

References

[1] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *ISSTA '06*, pages 169–180, New York, NY, USA, 2006. ACM Press.

[2] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. *ICSE*, 00:449, 2000.

[3] D. Saff. Theory-infected, or how I learned to stop worrying and love universal quantification. In *OOPSLA '07*, 2007.

[4] N. Tillmann and W. Schulte. Parameterized unit tests. *SIGSOFT Softw. Eng. Notes*, 30(5):253–262, 2005.

[3] http://junitfactory.org