Theory-infected

Or How I Learned to Stop Worrying and Love Universal Quantification

David Saff
MIT CSAIL
saff@mit.edu

Abstract

Writing developer tests as software is built can provide peace of mind. As the software grows, running the tests can prove that everything still works as the developer envisioned it. But what about the behavior the developer failed to envision? Although verifying a few well-picked scenarios is often enough, experienced developers know bugs can often lurk even in well-tested code, when correct but untested inputs provoke obviously wrong responses. This leads to worry.

We suggest writing *Theories* alongside developer tests, to specify desired universal behaviors. We will demonstrate how writing theories affects test-driven development, how new features in JUnit can verify theories against hand-picked inputs, and how a new tool, Theory Explorer, can search for new inputs, leading to a new, less worrysome approach to development.

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

Many developers write and frequently run automated tests to confirm that their code's behavior matches their intentions, reducing worry that they have misunderstood how their code works. Most popular developer testing frameworks support tests based on example scenarios. Thinking through examples can be a useful step toward clarifying a vague specification in the developers' head:

```
@Test sqrRootExamples() {
   assertEquals(2.0, sqrRoot(4));
   assertEquals(3.0, sqrRoot(9));
}
```


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

However, the developer's understanding of the unit's correct behavior will eventually grow beyond what can be expressed by individual examples. This leads back to worry: some unexpected future input might cause some future unexpected future implementation to produce behavior the developer knows right now is a bug, but the current tests won't catch it. The behavior of the unit is only defined in a few representative scenarios. Even future human maintainers may not be able to assume the correct generalizations: what is the right behavior when `sqrRoot` receieves a negative input? Zero? Inputs without integer square roots? Is a negative square root ever allowed?

To allow developers to eliminate this worry, we propose that developer testing frameworks should support, as first-order constructs alongside traditional tests, partial specifications over large or infinite sets of values, called *Theories*. Theories look like test methods, but are universally quantified: all assertions must hold for any possible parameter values that pass the assumptions. For example, the 4.4 release of JUnit allows the following statement:

```
@Theory defnOfSquareRoot(double n) {
  assumeTrue(n >= 0);
  assertEquals(n, sqrRoot(n) * sqrRoot(n), 0.01);
  assertTrue(sqrRoot(n) >= 0);
}
```

The `assumeTrue` clause states the assumption of the theory: sqrRoot is intended to work on any non-negative double. If `sqrRoot` should also have defined behavior on negative inputs, this may be expressed in a separate theory.

2. Uses

Not everything to be said about a codebase should be said in a Theory. The concrete example executions found in tests are a valuable source of feedback and future reference for the implementation and design of the code. However, Theories are effective at capturing several different kinds of ideas:

- Identities and data conversion. For example:

```
@Theory xmlConservesData(DataSet d) {
  assertEquals(d, readFromXml(d.toXml()));
```

```
}
```

- Globally disallowed behavior. For example:

```
@Theory parseIsNeverNull(String textLine) {
   assertNotNull(parser.parse(textLine));
}
```

- Preserved data, that is data whose content is inconsequential to the test, but whose preserved identity is crucial. For example:

```
@Theory pointConstruction(int x, int y) {
  assertEquals(x, new Point(x, y).getX());
}
```

3. Related work

Developers who wish to make general statements about the correct operation of their code can add assertions to their implementation, sometimes using a built-in language construct like Java's `assert` or Eiffel's design-by-contract [2] preconditions and postconditions. These can be useful for representation invariants, which refer only to internal private state. However, Theories are more effective than assertions for expressing and exploring general statements of externally-visible behavior:

- Theories can safely mutate the state of the objects under test, which is of course too dangerous in in-line assertions.
- Theories are in a separate class, where they can be added, removed, executed, hypothesized, read, and explored without changing or cluttering the implementation code.
- Theories exert positive design pressures, similar to unit tests. If it is difficult to write a Theory about a class's correct external behavior, clients of the class will be more difficult to write and understand.

Our work depends on Tillman and Schulte, who were the first to investigate the use of Parameterized Unit Tests [5] as a tool for test generation. Our Theory code constructs are only superficially different from PUTs, but our tool support and suggested use are different:

1. Theories are written at the same time as tests.
2. Theories can be *evaluated* in a fast feedback cycle by standard test frameworks using developer-supplied input values.
3. Theories can be *explored* in a slower cycle, using automated tools to look for violating inputs.

Theories provide a simple way to use sophisticated tools to *explore* for complicated failure conditions. Tools for random testing [3] and model checking [6] can be given simple interfaces suitable for quick use by novice developers, if they are only used to look for obviously faulty conditions: deadlocks, in-line assertion violations, or uncaught exceptions. However, only a subset of all bugs in a program will manifest in one of these conditions–for the others, developers must often learn specialized logic languages or extension API's. Theories provice a much simpler route, by adapting faulty behavior in the unit under test (`sqrRoot(16)` returns `-4`) to a simpler error condition (an assertion violation).

4. Tools

To support Java developers using Theories, we have developed two different tools. For Theory *evaluation*, we have provided a test runner that (now included in JUnit 4.4 [1]), which uses all possible annotated `DataPoints` from the test class as inputs to the Theory:

```
@RunWith(Theories) public class SquareRoot {
  @DataPoint public double FOUR = 4.0;
  @DataPoint public double NINE = 9.0;

  @Theory defnOfSquareRoot(double n) { ... }
}
```

For Theory *exploration*, we have developed a separate tool, *Theory Explorer*. When invoked on a Theory, Theory Explorer analyzes the Theory code and the code under test, attempting to find inputs that will cause the Theory to fail. If a failing input is found, the developer indicates if it represents a bug (in which case Explorer adds the input as a data point) or a missing assumption (in which case Explorer prompts for the new assumption). By iterating through the developer's theories until no violating inputs are found, the Theories are made more precise, bugs are found, and the developers' confidence increases.

The input generator for Theory Explorer uses the free JUnit Factory web service [4]. JUnit Factory uses random testing, heuristics, and symbolic execution to find inputs that cover new code path, but a license is needed to operate on proprietary code.

We encourage all Java developers to download these new tools, provide feedback, and stop worrying.

References

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

[2] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[3] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE '07*, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[4] A. Software. Junit factory. `http://www.junitfactory.org`.

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

[6] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA '04*, pages 97–107, New York, NY, USA, 2004. ACM Press.