At the Intersection of Applied Formal Methods and Unit Testing

Daniel M. Zimmerman
Institute of Technology
University of Washington Tacoma

NTU Graduate Seminar, 台北 - 7 January 2011
Outline

• Applied Formal Methods
• Java and the Java Modeling Language
• Formal Contract the Design
• JMLUnitNG
Applied Formal Methods

• **formal methods** are mathematical techniques for building verifiably-correct software systems

• **applied formal methods** is the creation and evaluation of techniques and tools that make formal methods accessible and useful to developers who may not know all the mathematics involved
Correctness

- a **correct** software system is one that does what it’s supposed to
Correctness

• a **correct** software system is one that does what it’s supposed to

• correctness is always **relative**!
Correctness

• a **correct** software system is one that does what it’s supposed to

• correctness is always **relative**!

• you need a **specification** of what a system is supposed to do before you can evaluate its correctness
Specifications

• specifications of software range in formality:
  • informal - English documentation (e.g., “normal” comments in code)
  • semi-formal - structured English documentation (e.g., Javadoc)
  • formal - annotations and assertions (e.g., assert statements and contracts)
Specifications

• specifications of software range in formality:
  • informal - English documentation (e.g., “normal” comments in code)
  • semi-formal - structured English documentation (e.g., Javadoc)
  • formal - annotations and assertions (e.g., assert statements and contracts)
• most developers do the first, some do the second, not too many do the third
/** Debit this account. * @param amount the amount to debit. * @result the resulting balance. */

/*@ requires 0 <= amount && amount <= balance; @ ensures balance == \old(balance - amount) && @ \result == balance; @*/

public int debit(int amount)
Design by Contract

- **contracts** are a **key concept** in robust software design and construction

- **precondition**: an assertion that must be true before a method can be called

- **postcondition**: an assertion that is guaranteed to be true when a method returns

- **invariant**: an assertion that is true of an object in observable states
Design by Contract

- in a Design by Contract process, the contracts for all the classes and methods are written first
- once all the contracts are written, the code is written to “fill in the blanks”
- the contracts serve as design documentation (hence the name)
Java Modeling Language

• the contracts we just saw were written in the Java Modeling Language (JML), a notation for formally specifying the behavior and interface of Java classes and methods

• originally developed by Gary T. Leavens (Iowa State, now U. Central Florida) and others, now worked on by researchers worldwide (including me)

• many tools understand JML, including runtime checkers and static verifiers
Runtime Checking

- runtime checking is the process of evaluating preconditions, postconditions, invariants, and other assertions at runtime.

- If an assertion fails at any point, an exception/error occurs and execution halts (hopefully with a useful message about what happened).

- jml4c is a runtime checking compiler for JML that supports modern Java syntax.
Static Verification

• *static verification* is the process of checking, using automated theorem proving and similar mechanisms, that specifications written in a language such as JML are satisfied by the corresponding code

• *ESC/Java2* is a static verifier for JML that can handle a wide range of possible JML specifications
Using JML to Specify Java Programs

• we can use JML to specify many things about the behavior of Java programs, up to full logical models of program behavior

• we can prove the correctness of these specifications with static verifiers

• we can check the correctness of the program at runtime with runtime checkers
Using JML to Specify Java Programs

- this *should* enable us to write much more reliable Java programs
Using JML to Specify Java Programs

• this *should* enable us to write much more reliable Java programs

• the catch: Java has a *huge* (1000s of classes) standard library, and we need specs for these library classes before we can fully specify the behavior of programs that use them
Generating Specs for Library Classes

• generating correct specs for the library classes is relatively easy - you can write a correct, but trivial, specification for anything!
Generating Specs for Library Classes

• generating correct specs for the library classes is relatively easy - you can write a correct, but trivial, specification for anything!

• precondition true, postcondition true, invariant true...
Generating Specs for Library Classes

• generating correct specs for the library classes is relatively easy - you can write a correct, but trivial, specification for anything!

• precondition \textit{true}, postcondition \textit{true}, invariant \textit{true}...

• generating good specs for the library classes that allow us to reason about programs - not too trivial, but also not too strict or too complex - is hard!
Generating Specs for Library Classes

- what we can work with:
  - Javadoc for the standard library
  - the suite of automated tests provided by Oracle in the Java Compatibility Kit (JCK), which must all pass in a “compliant” implementation of the standard library
  - working directly with the library source code is unwise, since it changes every release and across Java implementations
Better Specifications through Testing

• **idea**: why not use the comprehensive suite of tests for the standard library to check our specifications somehow?
Better Specifications through Testing

• **idea**: why not use the comprehensive suite of tests for the standard library to check our specifications somehow?

• if we can statically verify that the tests pass with our new library specs, then for all practical purposes the specs are good
Better Specifications through Testing

- **idea**: why not use the comprehensive suite of tests for the standard library to check our specifications somehow?

- if we can statically verify that the tests pass with our new library specs, then for all practical purposes the specs are good

- effectively, we test our specifications by verifying the existing tests
Verifying Unit Tests

- we assume that our unit test framework uses an “assert” method to check Boolean conditions and a “fail” method to trigger a failure without a condition check

- in order to statically verify unit tests, we add very simple specifications to these methods:
  - `assert(x)` has precondition `x` and postcondition `x`
  - `fail()` has precondition `false` and postcondition `true`
Verifying Unit Tests

• using these specifications, our unit tests can be statically verified as follows:
  
• calls to library methods are verified against the method specs we’ve written
  
• calls to `assert(x)` will verify properly if `x` is `true`, exactly the behavior we want
  
• calls to `fail` will never verify (precondition `false`), which is good since such calls should be unreachable in tests that pass
Formal Contract the Design

• the specification process based on this idea is “Formal Contract the Design” (FCTD)

• “Contract the Design” is the opposite of “Design by Contract” - writing the contracts for a program after the program has been written rather than before

• in this case, we are writing specs for library classes whose operation has already been informally documented
The FCTD Process for Class $C$

- write an initial JML spec for $C$, using *only* the Javadoc for $C$ and any classes on which $C$ depends (*not* $C$’s source code or tests)
The FCTD Process for Class $C$

- write an initial JML spec for $C$, using only the Javadoc for $C$ and any classes on which $C$ depends (not $C$’s source code or tests)

- refine the spec for $C$ until it statically verifies against $C$’s source code, without looking at the source code
The FCTD Process for Class C

• write an initial JML spec for C, using only the Javadoc for C and any classes on which C depends (not C’s source code or tests)

• refine the spec for C until it statically verifies against C’s source code, without looking at the source code

• refine the spec for C until all the JCK tests for C statically verify (looking at the test code is OK here) – note that the tests are only checked and never run!
Current Status

• using this process, we have specified several classes in the Java standard library so far - concentrating on commonly-used classes such as the Collections Framework

• obviously it will take significant effort to (re)specify the entire standard library, but it’s a lot easier when we can leverage the JCK to check our specs
Switching Gears...

- I’ve been talking about using existing unit tests to help generate JML specifications
Switching Gears...

• I’ve been talking about using existing unit tests to help generate JML specifications

• next, I’ll discuss using existing JML specifications to generate unit tests

• a version of this functionality has existed since early versions of JML, in the form of JMLUnit
JMLUnit: The Basic Idea

- JMLUnit works by using runtime assertion checking (RAC) code, generated from the JML specifications for methods and classes, as test oracles.
- The test data needs to be defined by the test developer, but all the test code is generated automatically.
JMLUnit: The Basic Idea

• each test calls one method with one set of data and has three possible outcomes:
  • success, if there were no assertion failures
  • failure, if there was a failure in an assertion other than the method’s precondition
  • meaningless, if there was a failure in the method’s precondition – because methods can do anything if their preconditions are violated (so there’s no way to “fail”)

JMLUnit Shortcomings

- JMLUnit is easy to use and understand, but has some shortcomings:
  - it requires developers to manually specify test data (at least instances to test), often in a less-than-obvious way
  - it ignores context, using the same data set for each parameter of the same type
  - it can easily consume extreme amounts of memory (run for weeks with no results!)
JMLUnitNG

• since JML is being modernized, we felt it was time to both modernize JMLUnit and address these shortcomings

• we wanted to keep the principle of operation easy for first-time JML users to understand, rather than to be the best testing tool in existence
New Groundwork: TestNG

- JUnit was the only testing framework for Java when JMLUnit was written - TestNG came later and added nice features
- parameterized tests can be specified in a way that allows lazy generation of test data sets at runtime
- the concept of a skipped test is built into the framework
- (bonus!) parallel testing is trivial to enable
Improvement: Memory Usage

- TestNG’s parameterized testing allows us to completely eliminate the excessive memory usage of JMLUnit
- instead of constructing all parameter lists at once and storing them in memory, we use special data generation iterators to generate parameter lists on-the-fly, as needed
- we can easily run millions of tests
Improvement:
Test Data Specification

- JMLUnitNG allows developers to easily specify additional test data, including context-sensitive data
public class Add {
    //@ invariant x() + y() > 0;

    private int my_x;
    private int my_y;

    //@ requires the_x + the_y > 0;
    //@ ensures x() == the_x && y() == the_y;
    public Add(final int the_x, final int the_y) {
        my_x = the_x;
        my_y = the_y;
    }

    public /*@ pure @*/ int x() { return my_x; }
    public /*@ pure @*/ int y() { return my_y; }

    //@ ensures \result == x() + y() + the_operand;
    public /*@ pure @*/ int sum(final int the_operand) {
        return my_x + my_y + the_operand;
    }
}
Test Data Specification
The Old Way

• running JMLUnit creates 2 Java classes

• one is test fixtures and gets left alone

• the other is test data - it’s 162 lines long, and the two parts we need to edit to add new test data are on lines 122 and 157

• if we want specific data values to be used in specific places, we have to manually add new logic to the test data class
Test Data Specification
The New Way

• running JMLUnitNG creates 6 Java classes
  • one is test fixtures and still gets left alone
  • the others are context-sensitive test data - one for each method parameter (2 for the constructor, 1 for “sum”), one for each type (int), and one for Add itself
  • to change the data used in a particular context, we change the appropriate class
Test Data Specification
The New Way

- each generated test data class looks like this:

```java
/**
 * Test data strategy for Add. Provides test values for
 * parameter "int the_operand" of method "int sum(int)".
 *
 * @author JMLUnitNG 1.0a2 (42)
 * @version 2011-01-06 00:18 +0800
 */
public class sum__int_the_operand__the_operand
    extends GlobalStrategy_int {
    /**
     * @return custom values for parameter "int the_operand".
     */
    public RepeatedAccessIterator<?> getCustomValues() {
        return new ObjectArrayIterator<Integer>(new Integer[] { /* add custom int values here */ });
    }
}
```
Improvement: Test Object Generation

- JMLUnit tests constructors, but nothing else, if you just run its generated tests with no editing - test objects must be supplied manually
- JMLUnitNG uses Java reflection to instantiate test objects with the parameter lists from successful constructor tests
- We can test three Add objects with no developer intervention whatsoever
public class Add {

    //@ invariant x() + y() > 0;

    private int my_x;
    private int my_y;

    //@ requires the_x + the_y > 0;
    //@ ensures x() == the_x && y() == the_y;
    public Add(final int the_x, final int the_y) {
        my_x = the_x;
        my_y = the_y;
    }

    public /*@ pure @*/ int x() { return my_x; }
    public /*@ pure @*/ int y() { return my_y; }

    //@ ensures \result == x() + y() + the_operand;
    public /*@ pure @*/ int sum(final int the_operand) {
        return my_x + my_y + the_operand;
    }
}
Improvement: Test Object Generation

- we use reflection in a similar way to generate objects for use as parameters to test methods
- by manually adding a few well-chosen primitive values to the defaults, more objects are reflectively created
Results

• when run on some examples, we experienced significant increases in “hands-off” test coverage over the original JMLUnit

• we experienced much larger increases when adding a few additional data values for use in specific contexts
Current Status

• JMLUnitNG is publicly available at http://formalmethods.insttech.washington.edu/ - current version is 1.0 alpha 2

• it works, but is not yet as user-friendly as it should be (no Eclipse plugin, for example)

• currently requires the use of the jml4c compiler to work with Java >= 1.5.
Quick Demo

- into Eclipse we go!
Summary

- two projects that combine unit testing and applied formal methods:
  - Formal Contract the Design
  - JMLUnitNG
- work is ongoing on both, with many improvements in the works for JMLUnitNG