Testing Library Specifications by Verifying Conformance Tests

Joseph R. Kiniry, Daniel M. Zimmerman, Ralph Hyland
ITU Copenhagen, UW Tacoma, UCD Dublin

6th International Conference on Tests & Proofs
Prague, Czech Republic - 31 May 2012

The Main Idea

- good library specifications are essential for modular verification
- many libraries have no specifications, but good conformance tests
- we can use conformance tests to ensure that post hoc library specifications are *correct* and *useful*

Motivation: JML and Static Checking

- behavior of Java programs can be specified with Java Modeling Language (JML)
- modular verification can be performed with extended static checkers like ESC/Java2
- we need good specifications for classes not being checked/verified!

Motivation: Java Class Library

- the Java class library is *huge* (1000s of classes in over 100 packages)
- it has no formal specs
- its documentation is primarily informal English in Javadoc comments

Motivation: Java Class Library

- we want *correct* specs for the Java class library - but correct isn't enough
  - precondition *true*, postcondition *true*, invariant *true*...
- we also want *useful* specs, so we can actually verify nontrivial programs against them

Motivation: Java Class Library

- Java 1.4 library specs shipped with JML2 were hand-written as needed, in ad hoc fashion, over several years
- their correctness has primarily been a matter of trial and error
- no way to measure their utility other than by attempting to verify programs

Example: *java.lang.Byte*

```
public /*@ pure @*/ final class Byte extends Number implements Comparable
{

    //@ public model byte theByte;
    //@ represents theByte <- byteValue();

    /*@
      @    public normal_behavior
      @      requires Character.MIN_RADIX <= r && r <= Character.MAX_RADIX;
      @   assignable \nothing;
      @      ensures \result <==>
      @              s != null && !s.equals("") &&
      @              (\forall int i; 0 <= i && i < s.length();
      @                      Character.digit(s.charAt(i), r) != -1);
      @    also
      @    public normal_behavior
      @   requires Character.MIN_RADIX <= r && r <= Character.MAX_RADIX;
      @   assignable \nothing;
      @   ensures \result <==>
      @              s != null && !s.equals("") &&
```

And another 200 lines after that, for methods of *java.lang.Byte*!

Better Specifications through Testing

- **idea:** use the conformance test suite for the Java class library – the Java Compatibility Kit (JCK) – to evaluate library specifications

Better Specifications through Testing

- the JCK tests are *operational specifications* for the behavior of the Java class library
- they should be statically verifiable against post hoc JML specifications
- effectively, we can *test* our *specifications* by *verifying* the existing *tests*

Verifying Unit Tests

- we assume a unit test framework with 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:
  - {*x*} assert(*x*) {*x*}
  - {false} fail() {true}

Verifying Unit Tests

- unit tests can then be statically verified as follows:
  - calls to library methods are verified against the library specs as necessary
  - calls to *assert(x)* will verify properly if *x* is *true*, exactly the desired behavior
  - calls to *fail* will never verify (precondition *false*) – but such calls are unreachable in tests that pass

Formal Contract the Design

- the specification process based on this idea is called *Formal Contract the Design* (FCTD)
- *Contract the Design* is the dual of *Design by Contract* – writing contracts for a program *after* the program has been written
- in FCTD, contracts are written for classes with informal documentation and unit tests are used to validate them

The FCTD Process (Java/JML) for Class *C*

- write an initial JML spec for *C*, using *only* Javadoc for *C* and any classes on which *C* depends (*not* source code or JCK tests)
- refine the spec for *C* until it statically verifies against *C*'s source code, *without looking at C's source code*
- when the *C* spec can be statically verified against *C*, it is *correct*

The FCTD Process (Java/JML) for Class *C*

- attempt to statically verify the tests for *C* using the new spec – the tests are only *checked* and never *run*!
- define spec *utility* as the percentage of the tests for *C* that statically verify
- refine the *C* spec until its utility is 100% (making sure it remains correct!)
  - looking at test code to see what tests do and expect is OK, if necessary

Example - *java.util.String.getChars()*

```
public void getChars(int srcBegin, int srcEnd,
                     char[] dst, int dstBegin)
```

- copies characters from a string into a destination array
- various things can go wrong depending on the supplied parameters

Example - *java.util.String.getChars()*

- Javadoc for *getChars()* describes situations that cause *IndexOutOfBoundsException*, but does not mention *NullPointerException*
- original JML2 spec written for *getChars()* did not account for *NullPointerException*
- a JCK test (the very first one for *getChars()*!) checks for *NullPointerException*, so FCTD captures it even though it is undocumented

Current Status

- we have specified several classes in the Java standard library using FCTD, concentrating on commonly-used classes such as the Collections Framework
- obviously, it will take significant effort to (re)specify the entire Java standard library
- the process is a lot easier when we can leverage the JCK to check our specs

Broader Applicability

- FCTD is directly applicable to libraries with automated conformance tests in languages/runtimes with available modular static verification tools
- FCTD can also be used when performing CTD for non-library programs if high-coverage, high-quality unit tests are available

Conclusion

- *Formal Contract the Design* allows us to use existing *operational* specifications to evaluate new *denotational* specifications
- currently being used to develop the next generation of Java class library specifications
- future: integration with specification inference methods, integration with test generation methods that don't use specs, other ways to measure spec utility