Testing Library Specifications by Verifying Conformance Tests

Kiniry, Joseph; Zimmerman, Daniel M.; Hyland, Ralph

Published in:
Tests and Proofs

Link to article, DOI:
10.1007/978-3-642-30473-6_6

Publication date:
2012

Testing Library Specifications by Verifying Conformance Tests

Joseph R. Kiniry[1], Daniel M. Zimmerman[2], and Ralph Hyland[3]

[1] IT University of Copenhagen, Denmark, kiniry@acm.org
[2] University of Washington Tacoma, USA, dmz@acm.org
[3] University College Dublin, Ireland, ralph.hyland@gmail.com

Abstract. Formal specifications of standard libraries are necessary when statically verifying software that uses those libraries. Library specifications must be both *correct*, accurately reflecting library behavior, and *useful*, describing library behavior in sufficient detail to allow static verification of client programs. Specification and verification researchers regularly face the question of whether the library specifications we use are correct and useful, and we have collectively provided no good answers. Over the past few years we have created and refined a software engineering process, which we call the *Formal CTD Process* (FCTD), to address this problem. Although FCTD is primarily targeted toward those who write Java libraries (or specifications for existing Java libraries) using the Java Modeling Language (JML), its techniques are broadly applicable. The key to FCTD is its novel usage of library conformance test suites. Rather than executing the conformance tests, FCTD uses them to measure the *correctness* and *utility* of specifications through static verification. FCTD is beginning to see significant use within the JML community and is the cornerstone process of the JML Spec-a-thons, meetings that bring JML researchers and practitioners together for intensive specification writing sessions. This article describes the Formal CTD Process, its use in small case studies, and its broad application to the standard Java class library.

1 Introduction

In an ideal world, all software systems would be 100% reliable and have verifiable formal specifications. These specifications would be both *correct*, accurately reflecting the runtime behavior of the implementations they claim to describe, and *useful*, providing sufficient information to allow developers to use and extend the implementations in safe, behaviorally predictable ways.

Clearly, we do not—and will likely never—live in this ideal world. The vast majority of today's software has defects, and some is completely unreliable. Of the software that is considered robust and reliable, most has not undergone formal verification. A commonly discussed way in which the reliability of such software is determined is through exhaustive automated unit testing, both of individual components and of full systems. Very little of today's software has

formal specifications of any kind and, in our experience, many of the formal specifications that do exist are either incorrect or not useful.

While it is clearly impossible to achieve an ideal, 100% reliable, fully-specified world, it is certainly possible to improve the current state of software specification and reliability. One approach to doing this for Java applications, taken by the Java Modeling Language (JML) community, has been to retroactively provide formal specifications for the behavior of the standard Java class library. Since the standard library is the foundation for all Java software, this enables Java programs to be formally specified and verified using the standard library specifications as building blocks. A set of specifications for much of the Java 1.4 class library was included as part of the release of the Common JML tool suite [2], and has since been used to formally specify and verify multiple large Java systems [13,14]. In addition, a set of specifications for Java 1.5 through 1.7 is under development as part of the OpenJML project.

We call such retroactive specification of classes *Contract the Design* (CTD). To perform CTD, the specification writer takes an implementation that already exists, such as the Java class library, and writes *contracts* (formal specifications) to describe the existing behavior. This approach is effectively the logical dual of *Design by Contract* (DBC) [17], where contracts are written first and the software is subsequently implemented in a way that fulfills the contracts.

Unfortunately, two critical difficulties arise when using CTD to specify existing systems. The first is that retroactively devising good specifications for existing systems is quite difficult, even with full access to source code, and is especially problematic when documentation is incomplete, absent, or vague (a not infrequent occurrence). The second is that, once a specification has been written for a non-trivial piece of software, it is hard to determine whether the specification is correct and useful. Imagining all the situations in which the software might be used is infeasible, and it is impossible to just "try out" the specification with the multitude of tools that will consume and use it. These difficulties are also faced by developers writing conformance tests for libraries (and, more generally, unit tests for nontrivial software systems), who must devise sets of tests that thoroughly exercise the functionality of their systems.

As a result of these difficulties, the Java class library specifications currently packaged with JML were written over several years in essentially ad hoc fashion. Efforts were made to ensure their correctness, and they are all (to the best of our knowledge) at least type-correct; however, there was no systematic way to measure their utility and, until now, no attempt to devise such. Deficiencies in the specifications have primarily been discovered by application developers attempting to verify their code and by tool developers attempting to make their tools understand the specifications, and the specifications have been patched in various ways over time in response to these discoveries.

This work addresses the difficulties in using CTD for library specification, taking advantage of the substantial body of knowledge related to the creation of high quality unit test suites by combining existing testing techniques with runtime and static verification techniques. The resulting process, which we call

the *Formal CTD Process* (FCTD), allows us to effectively test specifications for correctness and measure their utility.

The *Formal* in *Formal CTD Process* refers to the fact that we use verification tools to determine the *correctness* and *utility* of specifications. We consider the correctness of a specification to be a binary property: a specification is correct if it is never violated by the implementation it claims to specify and incorrect otherwise. This can be determined statically or, in cases where the source code of the system being specified is unavailable, evidence for such can be observed dynamically. We measure the utility of a specification as the percentage of the unit tests for the specified implementation that can be statically verified using the specification.

The "secret sauce" of this process is the realization that unit tests and verification "fit" together well. Unit tests *operationally* express the correct behavior of a system under test (SUT), while specifications *denotationally* express the correct behavior of a SUT. Thus, unit test outcomes *should* be statically verifiable using SUT specifications. Just as *test coverage* criteria tell us something about the quality of our *implementations*, *verification coverage* criteria (correctness and utility) tell us something about the quality of our *specifications*.

FCTD has been used in several case studies, and we have devised a method to apply it to the entire standard Java class library. The resulting ability to test the specifications of the Java class library is playing a critical role in the effort to update JML to support current and future versions of Java. The end result of this effort will be a correct and useful JML specification of the Java class library to complement the (often poor) existing Javadoc "specification", which will both clarify the behavior of the library for application developers and allow them to confidently use the many verification and validation tools that understand JML. In the remainder of this article we provide background information about the components of the FCTD process, describe the principles underlying FCTD, and discuss concrete realizations of FCTD for JML and Java code.

2 Background

FCTD combines unit testing and verification to evaluate specifications. In this section, we provide background information on the tools and techniques used in our concrete realizations of FCTD: JML, unit testing, and static checking. More detailed descriptions are available in the cited works and in the documentation accompanying the tools.

2.1 The Java Modeling Language

The Java Modeling Language [16] is a specification language for Java programs. It allows developers to specify class and method contracts (i.e., preconditions, postconditions, invariants) as well as more sophisticated properties, up to and including mathematical models of program behavior. Many tools are compatible

with JML, including compilers, static checkers, runtime assertion checkers, unit test generators, and automated specification generators [2].

Runtime assertion checking (RAC) is one of the most common applications of JML. A special RAC compiler is used to transform JML-annotated Java code, adding runtime checks for method preconditions and postconditions, class invariants and constraints, and other JML contracts such as loop invariants and variant functions. The transformed code, when run in any Java virtual machine, signals assertion failures at runtime via special JML runtime errors.

JML and its associated tools are used in many contexts. Many formal methods researchers have adopted JML, and many JML-compatible tools have full JML specifications; in some cases, these tools have even been statically verified. JML is also used in educational contexts at several universities [15], as well as in production systems such as the KOA Internet-based remote voting system [13].

Currently, most stable and supported publicly available JML tools only support Java versions prior to 5.0. The OpenJML project [5] and the JMLEclipse project [3] are efforts to build JML tools atop modern compiler code bases (OpenJDK [19] and the Eclipse JDT [22], respectively), so that JML will be able to better keep pace with future changes to Java. This modernization effort is a primary motivator for the work presented here, as a multitude of new specifications must be written for new and changed classes in the standard Java class library before a new set of JML tools can be effectively used.

2.2 Unit Testing

Unit testing has long been an important validation technique in many software development processes. It is essentially the execution of individual components of a system (the *units*) in specific contexts to determine whether the components generate expected results. A single *unit test* has two main parts: *test data* and a *test oracle*. The *test data* are input values—for example, method parameter values—used to establish the state of the unit under test. The *test oracle* is a (typically small) program that determines whether the behavior of the unit is correct when it is executed with the test data.

Testing a non-trivial software system typically requires many unit tests, which are collectively called a *test suite*. The quality, usually called *coverage*, of a test suite is measured in several ways [23]; for example, given a particular SUT, *statement coverage* (sometimes called *code coverage*) is the percentage of the executable code in that system that is executed during testing, while *path coverage* is the percentage of the possible execution paths through that system that is executed during testing.

As will become evident in Section 3, effective use of FCTD requires the availability of a high-coverage test suite for the system being specified, preferably a *conformance test* suite that defines the full functionality of the system. Significant research has been done on techniques for automatically generating test suites, and many of those techniques are quite promising. However, the development of test suites is still predominantly done without automation. Developers sit down with the system to be tested, decide what test data should be used

and how to determine whether each test has passed or failed, and encode this information manually.

Test suites are usually executed within a test framework that runs and records the result of each test and summarizes the results of the suite to the developer. This allows the test suite to be run easily, repeatably, and without constant developer supervision; it also allows test runs to be incorporated into the automatic build processes that exist in many software development environments. Such frameworks exist for nearly every programming language; the predominant ones for Java are JUnit [10] and TestNG [1]. For the purposes of FCTD there is no clear reason to choose one over the other; we have typically used JUnit because it ships as an integrated part of the open-source Eclipse Development Platform [22], which we use for most of our development work.

One particular way of automating the development and execution of test suites that applies specifically to JML-annotated Java programs is embodied in the JMLUnit [4] and JMLUnitNG [24] tools. These tools automatically generate test oracles using JML runtime assertion checks. For each method under test, the tools construct JUnit or TestNG (respectively) tests to call that method with multiple test data values taken from a default or developer-provided set. Each test passes if the method call completes with no assertion failures and fails if the method call completes with an assertion failure other than a violation of the method's precondition. If a test violates the method's precondition it is considered *meaningless*, because the behavior of a method is undefined when its precondition is violated and can therefore not be evaluated.

FCTD does not work with unit tests generated by tools like JMLUnit and JMLUnitNG, because such tools generate operational "specifications" (in the form of unit tests) *directly from* denotational specifications (in the form of JML). Since the resulting tests pass exactly when the specifications are satisfied, regardless of how trivial the specifications are, they can provide no information about specification utility. To guarantee "objective" evaluation of specification utility, unit tests must be written *independently* (i.e., without reference to the specifications being evaluated).

2.3 Static Verification

Static verification is a process whereby a body of formally specified source or object code is analyzed to determine whether it satisfies its specification. This analysis is typically carried out by transforming the code and specification into verification conditions, which are then evaluated using one or more automated theorem provers. An *extended static checker* is a tool that performs static verification as well as checking for and flagging common programming errors.

In our initial experiments with FCTD we have primarily used ESC/Java2 [14], an evolution of the original Digital SRC ESC/Java [8], to perform static verification on JML-annotated Java code. We have also experimented with the prototype ESCs being developed as part of the OpenJML and JMLEclipse projects, but these are not currently robust enough to reason about the rich specifications under discussion here.

ESC/Java2 detects typical Java programming errors such as null pointer dereferences, invalid class casts, and out-of-bounds array indexing. It also performs several kinds of automated verification to attempt to ensure that the code is correct with respect to its associated JML specifications and, in conjunction with a specification consistency checker, to ensure that the specifications themselves are sound. The consistency check is important, especially for specification writers who are just learning CTD/DBC, because it is easy for inexperienced developers to write inconsistent specifications (e.g., invariants that collapse to `true`) that are always satisfied regardless of the system's actual behavior.

The verification performed by ESC/Java2 is *modular*; it verifies each method m by transforming the Java code of only method m into verification conditions and relies on the JML specifications of all the other methods and classes to which m refers to create the remainder of the verification conditions it needs. Thus, ESC/Java2 verifies each method in relative isolation, which is far less resource-intensive (and far more feasible given the state of automated theorem prover technology) than processing an entire system, or even an entire class, at once.

Support for modern Java syntax and constructs in ESC, as in JML itself, is still a work in progress. We expect that the new ESC tools being developed as part of the OpenJML and JMLEclipse projects will address this issue.

3 The Formal CTD Process

The Formal CTD Process is a combination of unit testing techniques and verification techniques. FCTD is general enough to be applied to systems and specifications written in any language and unit tests running in any framework; the only requirement is the availability of at least one static verification tool capable of reasoning about off-the-shelf unit tests.

It is important to note that executing unit tests is a *completely optional* part of the process. Executing unit tests within a RAC environment to determine whether CTD specifications are violated by the tests is certainly feasible; however, there are two serious issues with using runtime checks to evaluate specification quality. First, it is quite easy to write correct, but clearly non-useful, specifications that pass their runtime assertion checks regardless of what the underlying implementation does. Second, it is possible for all the runtime checks to pass when running a test suite that does not thoroughly exercise the implementation, leaving incorrect specifications undiscovered.

A conformance test suite (or other high-coverage test suite) can provide some persuasive evidence for the *correctness* of a set of specifications through runtime checking, because it thoroughly exercises all the intended functionality of the implementation. However, even runtime checks of conformance tests cannot *conclusively* establish correctness because there is always the possibility of undesirable "easter eggs" (e.g., "when a specific, undocumented set of parameters is passed to this method, exit the virtual machine") in any given implementation. To truly establish correctness, the specification must be statically verified against the implementation; if source code is not available, the best we can do is to per-

form runtime checking of a conformance test suite and hope (or, when possible, measure with coverage tools) that it completely covers the implementation.

On the other hand, it is impossible to determine the *utility* of a specification through runtime checking regardless of the test suite used. Runtime checking provides no basis for determining whether a specification will enable us to prove the correctness of programs that use the specified system.

In this section, we describe how FCTD allows us to determine the utility of specifications; in the next section, we describe our specific implementations of FCTD for testing JML specifications of Java systems.

3.1 Unit Tests as Operational Behavioral Specifications

As noted above, CTD specifications written for an existing system are *correct* if they can be statically verified. The use of static verification to determine specification correctness is one aspect of FCTD, but the critical hypothesis underlying the process is the following: *if correct CTD specifications for a system are sufficient to allow an existing high-quality and high-coverage test suite for the system to be statically verified, then for all practical purposes the specifications are useful.* By statically verifying a full test suite, we effectively "test" the ability of the specifications to capture the *intended* behavior of the system. The existing test suite is an operational behavioral specification of the system, against which we compare our formal specifications.

To illustrate this idea, consider the method `java.lang.String.getChars` from the Java class library. Its signature and Javadoc documentation (Figure 1) are fairly straightforward. A careful reading of this documentation (plus the knowledge that, in Java, performing array operations on a `null` array causes a `NullPointerException`) yields a 23-line JML specification (Figure 2) with three behavior clauses, one normal and two exceptional. This specification refers to `charArray`, a model field representing the character sequence encapsulated by the `String`, and `equal`, a model method that compares ranges of characters in two arrays for equivalence; both of these are inherited from `java.lang.CharSequence`, an interface implemented by `String`.[4]

Surprisingly, the reference implementation of `getChars` is only 7 statements long. Three `if` statements check the legitimacy of parameters and, if necessary, throw various instances of `StringIndexOutOfBoundsException`. Once the parameters are found to be legitimate, a single call to `System.arraycopy` copies the characters from the string into the destination array.

Verifying the correctness of this specification with respect to the reference implementation is a straightforward proposition—after all, the body of the method is short and, while it has a relatively high cyclomatic complexity given its size, its "shape" matches that of the specification. Moreover, the specification of `System.arraycopy` is strong and well-used.

[4] The specification would, of course, be significantly longer if it did not use the inherited model field and model method.

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

Copies characters from this string into the destination character array. The first character to be copied is at index `srcBegin`; the last character to be copied is at index `srcEnd-1` (thus the total number of characters to be copied is `srcEnd-srcBegin`). The characters are copied into the subarray of `dst` starting at index `dstBegin` and ending at index `dstbegin+(srcEnd-srcBegin)-1`.

(parameter descriptions omitted; they contain no restrictions on parameter values)

Throws `IndexOutOfBoundsException` if any of the following is true: `srcBegin` is negative; `srcBegin` is greater than `srcEnd`; `srcEnd` is greater than the length of this string; `dstBegin` is negative; `dstBegin+(srcEnd-srcBegin)` is larger than `dst.length`.

Fig. 1. Javadoc documentation for library method `java.lang.String.getChars`.

```
  public normal_behavior
    requires srcBegin >= 0
          && srcBegin <= srcEnd
          && srcEnd <= charArray.length
          && dst != null
          && dstBegin >= 0
          && dst.length >= dstBegin + (srcEnd - srcBegin);
    modifies dst[dstBegin .. dstBegin+srcEnd-srcBegin-1];
    ensures equal(charArray, srcBegin, dst, dstBegin, srcEnd - srcBegin);
also
  public exceptional_behavior
    requires srcBegin < 0
          || srcBegin > srcEnd
          || srcEnd > charArray.length
          || dstBegin < 0
          || (dst != null && dst.length < dstBegin + (srcEnd - srcBegin));
    modifies \nothing;
    signals_only IndexOutOfBoundsException;
also
  public exceptional_behavior
    requires dst == null;
    modifies \nothing;
    signals_only NullPointerException;
```

Fig. 2. JML specification for library method `java.lang.String.getChars`.

But how do we measure the utility of our 23-line `getChars` specification? To accomplish this, we turn our attention to the conformance test suite for the Java class library, the Java Compatibility Kit (JCK), which includes tests for `getChars` in the class `GetCharsTest`. This class is around 450 lines in length and contains nine comprehensive tests that call `getChars` a total of 36 times.

If we *execute* this test suite against the reference implementation of `getChars`, we exercise the implementation against the idea of correctness that the test writers had in mind when writing the tests. If, on the other hand, we attempt to *prove* that the unit tests always pass, we exercise our *specification* of `getChars` against that same idea of correctness.

One potential issue with this technique is *overspecification*—generating specifications detailed enough to prove that all the unit tests pass but too cumbersome

or too highly specific to the particular tested situations for developers to use in general applications. We attempt to avoid overspecification in two ways: first, we develop specifications only from publicly-available documentation (including test suites when necessary to clarify ambiguous documentation) and not directly from source code; second, we use conformance tests or other high-coverage, high-quality test suites in an effort to cover enough functionality that "specifying to the tests", if it is done at all, actually results in useful specifications.

We return to the JCK and `getChars` in Section 4.2; first, however, we explain how we combine unit testing and static verification to prove that the unit tests always pass.

3.2 Unit Test Specifications

In order to statically verify a test T, we must have specifications for both T itself and the test framework F against which T is written. Otherwise, T would be trivially verifiable as it need not maintain or establish any particular properties.

The specification for each test T is simple, and happens to match the default specification that ESC/Java2 assigns to any method that has no specification. Its precondition is `true`, as there are no constraints on when the unit test can be run;[5] its postcondition is also `true`, as we expect the test to terminate normally. Additionally, no exceptions should be thrown, so its *exceptional* postcondition is `false`. In essence, such a specification says that all unit tests should pass, no unit tests should fail and the test suite should not halt unexpectedly.

While test frameworks typically have a very rich set of methods to assert test conditions, we presume that test framework F contains only two methods—`Assert(boolean P)` and `Fail()`—and that all other assertion methods in F are defined in terms of these two methods. The `Assert` method does nothing if `P` is `true` and reports a test failure if `P` is `false`, while the `Fail` method unconditionally reports a test failure.

Framework F records a passing result for test T if T terminates normally. T may call the `Assert` method an arbitrary number of times to check the validity of an arbitrary number of predicates; if any of these assertions fail, F records a failing result for T. F also, of course, records a failing result for T if T calls the `Fail` method.

The specification of `Assert(P)` is {`P`} `Assert(P)` {`P`}. This specification forces the verifier to check that `P` holds, as it is the precondition of `Assert`. This is logically equivalent to simply asserting `P`. In case the test calls other methods of the framework API in the same method body, we also require that `P` holds as the postcondition.

The specification of `Fail` is {`false`} `Fail` {`true`}. This specification forces the verifier to attempt to prove an unprovable verification condition, and thus will always fail verification. The idea here is that the verification systems we use

[5] Unit tests that depend on the results of other unit tests, which can be written in certain testing frameworks, have somewhat more complex preconditions; however, such frameworks typically handle the ordering of such tests automatically.

have the ability to reason about unreachable code, and one way of checking for such is to attempt to assert `false` in unreachable blocks [12]. Since unit tests should never fail, all calls to `Fail` should be unreachable during static analysis.

With this combination of automatic default specification for the unit test and novel specification of test framework assertions, successful verification of a given unit test means that the unit test always passes: neither `Assert(false)` nor `Fail` is ever called. Therefore, it demonstrates that the CTD specification is sufficient to guarantee the success of the unit test. We measure specification utility as the percentage of unit tests in a test suite that can be statically verified; clearly, the correspondence between this measure of utility and actual "real-world" utility for developers is highly dependent on the quality of the test suite we verify. In particular, successful verification of a library conformance test—effectively, a complete definition of the required library behavior—against a library specification means that generalized client programs of the library should also be verifiable against the library specification (though they may still, of course, be unverifiable for reasons having nothing to do with the library specification).

Now that we have described the general idea of using verification to test the correctness and utility of specifications, we will discuss our concrete applications of this general idea using Java, JML, ESC/Java2, and Java unit testing frameworks.

4 The Concrete Process

We have been using the FCTD process for several years when writing new, or reworking existing, specifications of the Java API. An early variant of the concrete process was proposed by David Cok during the development of ESC/Java2. That process, which only used hand-written unit tests, the ESC/Java2 static checker and the JUnit framework, was used to write specifications of several classes. More recently one of the authors focused on refining the process, incorporating support for both static and runtime checking and using tests from the Java Compatibility Kit (JCK) [11].

In this section, we describe the modifications we have made to use JUnit and the JCK with the FCTD process, as well as giving background information on the organization of the JCK and its companion JavaTest framework.

4.1 JUnit

The FCTD process was originally designed for use with JUnit-like frameworks; therefore, the JML specifications added to the JUnit testing framework in order to successfully carry out FCTD are essentially those discussed in Section 3.2.

The JUnit API's key class, `org.junit.Assert`, has many methods that allow the test writer to assert various conditions. The most important two such methods are `assertTrue(String message, boolean condition)`, which asserts that `condition` is `true`, and `fail(String message)`, which causes an unconditional test failure. Figure 3 shows the JML specifications added to those

```
/** Asserts that a condition is true. */
//@ public normal_behavior
//@   requires condition;
//@   ensures condition;
public static void assertTrue(/*@ nullable @*/ String message,
                              boolean condition);

/** Fails a test with no message. */
//@ public normal_behavior
//@   requires false;
//@   ensures true;
public static void fail(/*@ nullable @*/ String message);
```

Fig. 3. The specification of key methods in JUnit's `Assert` class

methods for FCTD. A number of other methods, including `assertFalse` and `assertNull`, are implemented in terms of `assertTrue`; still others, such as `assertEquals` (for object equivalence) have their own implementations and we do not show the JML specifications for these here.

With these modifications to the JUnit API, we can perform FCTD with any existing set of JUnit tests for a JML-annotated SUT. We have sets of hand-written JUnit tests for a small selection of classes in the Java class library, as described later in Section 5; however, the conformance tests in the Java Compatibility Kit are preferable to our JUnit tests for testing the Java class library.

4.2 The Java Compatibility Kit

An implementation of the Java class library typically consists of a combination of Java code and native libraries. There are many such implementations, even on the same hardware and OS platform; Sun themselves implemented the class library for three platforms (Solaris, Windows, Linux), Apple implemented it for Mac OS and Mac OS X, and other implementations have been developed by companies such as IBM and Hewlett-Packard and open-source groups such as the Apache Harmony [21] and GNU Classpath [9] projects. In addition, OpenJDK [19], an open-source (GNU GPLv2+Classpath) version of Java first made available by Sun in 2007 and currently maintained by Oracle, runs on many different hardware and operating system platforms.

To ensure that the multitude of Java class library implementations would be mutually compatible and conform to the Java standard, Sun developed an extensive conformance test suite called the Java Compatibility Kit (JCK). Java licensees are required to ensure that their implementations pass the JCK tests before they can use the Java trademark.

Initially, Sun released the JCK to the public with a *read-only* license; the source code for the JCK was publicly available, but developers were explicitly permitted only to read the source code and not to compile or execute it.[6] The

[6] Sun's (now Oracle's) licensees, of course, have always had full access to the JCK under more liberal license terms.

```
//@ public normal_behavior
//@   ensures \fresh(\result);
//@   ensures \result.isPassed();
//@   ensures stringEquals(\result.getReason(), reason);
public static /*@ pure non_null @*/ Status passed
(/*@ nullable @*/ String reason);

//@ public normal_behavior
//@   requires false;
//@   ensures \fresh(\result);
//@   ensures \result.isFailed();
//@   ensures stringEquals(\result.getReason(), reason);
public static /*@ pure non_null @*/ Status failed
(/*@ nullable @*/ String reason);
```

Fig. 4. The specification of key methods in JavaTest's `Status` class

read-only license was one of the inspirations for the FCTD process, as it led us to consider ways in which we could make use of this conformance test suite while neither compiling nor executing it. Since the release of OpenJDK, however, Sun (now Oracle) has licensed the JCK for use by developers who are running it in conjunction with OpenJDK development or with projects that derive substantially from OpenJDK (such as the OpenJML project) [20].

JavaTest The JCK test suite is designed to be executed within a framework called JavaTest, which was also developed by Sun. The source for the JavaTest framework is available separately from the JCK distribution [18] and is (primarily) released under the same open-source license as OpenJDK. Thus, we are able to modify the JavaTest framework in a way that allows us to attempt static verification of all the unit tests in the JCK (and, for that matter, any other test suites written for the JavaTest framework).

The JavaTest API differs significantly from most standard test APIs in that it does not enable test code to directly assert predicates. Instead, "status" objects are constructed to indicate whether a given test branch passed or failed in some fashion. Thus, there is no direct analogue to the `Assert` method in the JavaTest API. When a test branch determines that it has succeeded, it always constructs and returns a status object representing a "success". Similarly, when a test branch determines that it has failed, it always constructs and returns a status object representing a "failure".

Our modifications to the framework are primarily to the class `com.sun.javatest.Status`, which implements the "status" object and therefore encapsulates the result of a single test. `Status` contains factory methods for creating objects that indicate that a test has passed, failed, or caused an error, and these methods are used by the JCK tests to report their outcomes.

To use FCTD with the JCK, we added JML specifications to these factory methods in the manner described in Section 3.2; these appear in Figure 4.[7] Note

[7] `stringEquals` is shorthand for "the two strings are equivalent or are both `null`."

```
public Status String0061() {
  String testCaseID = "String0061";
  String s = "getChar Test"; //step Create a String
  char[] dst = null; //step Create a null reference
  try {
    s.getChars(1,3,dst,0); //step Try to get chars
  }
  catch (NullPointerException e) { //step Catch an exception
    return Status.passed( "OKAY" );
  }
  return Status.failed( testCaseID + " getChars failed" );
}
```

Fig. 5. The first JCK test method for `String.getChars`

that the `failed` factory has a precondition of `false`, while the `passed` factory need not have pre- or postconditions relating to an asserted predicate because it always indicates a passed test. We also added minimal specifications to some related classes to ensure that we did not cause any `NullPointerException`s or similar runtime issues.

JCK Example An example will help to clarify the usage of the JavaTest framework for running automated tests. Recall that we discussed the method `java.lang.String.getChars` in Section 3.1. The first JCK test method for `getChars` (also the 61st JCK test method for the `String` class), `String0061`, is replicated in Figure 5. In this test, the success case is the `return` in the `catch` block where a "passed" instance of `Status` is constructed; this particular test is checking to see that passing a `null` array to `getChars` correctly causes a `NullPointerException`. The failure case is the final `return` where a "failed" `Status` is constructed.

The JML specification of `getChars`, therefore, needs to be strong enough to guarantee that a `NullPointerException` is always thrown when `getChars` is called on a `String` and given a `null` destination array. If the specification is sufficient, `String0061` will pass its static verification. The JML specification in Figure 2 correctly guarantees a `NullPointerException` in this instance and allows `String0061` to be statically verified. Interestingly, the specification for `getChars` shipped with some versions of the JML tools (including the most recent version of the Common JML tools released in 2009) does *not* include the exceptional behavior clause with the `NullPointerException`; FCTD would have detected that deficiency by failing to statically verify `String0061` and reporting less than 100% utility for the `String` specification.

5 Case Studies

For our first case study, we wrote or rewrote specifications for 26 commonly-used classes in the Java class library: `AbstractList`, `ArrayList`, `Arrays`, `BitSet`,

`Boolean`, `ByteArrayInputStream`, `Character`, `Class`, `Collection`, `Comparable`, `Exception`, `File`, `InputStream`, `Integer`, `List`, `Long`, `Map`, `Math`, `Object`, `Properties`, `Set`, `String`, `StringBuffer`, `System`, `Throwable`, and `Vector`.[8] We used FCTD with hand-written JUnit tests to help ensure that the specifications were both correct and useful.

The second case study was conducted as a part of Hyland's MSc work [11]. Specifications for three classes that had no existing JML specifications were written and verified using the JCK tests. These classes—`ResourceBundle`, `PrintStream`, and `Stack`—were chosen because they are the most frequently used classes in the Java API (as measured by two static analysis tools) that were missing specifications.

In both case studies, JML specifications for a class C were written using Javadoc documentation for C and any classes of which C is a client. Reading publicly available unit tests was permitted, but not encouraged, and served only to resolve ambiguity in the documentation. In an effort to avoid overspecification, viewing C's source code was *not permitted* under any circumstances.

The struggles and outcomes of this process highlight the fact that natural language specifications, as seen in Javadoc documentation, are often imprecise and incomplete. This was witnessed both during attempts to verify the classes in question against their newly-written specifications and during attempts to test the specifications by verifying the JCK tests and other tests against them. Hyland's MSc report discusses in detail some of the challenges posed and solutions found while writing usable and correct specifications for these three classes.

During both case studies, a specification was deemed *correct* only if (a) a manual review by multiple parties of the informal documentation and the formal specification had a positive outcome, (b) hand-written unit tests all passed when executed with runtime assertion checking of the JML specifications turned on, and (c) ESC/Java2 verified all unit tests successfully. If unit tests could not be executed, as in the case of the JCK, then part (b) was not performed. The utility of a class specification was measured in these case studies as the ratio of the number of verified unit tests for the class to the total number of unit tests run for the class, and no class specification was declared "finished" until its utility was measured at 100%. All class specifications were declared "finished" by the end of the case studies.

As a result of these case studies, we have much higher confidence in the correctness and utility of the tested specifications than we did for specifications that were written in the past using more ad hoc techniques. Consequently, the JML community is continuing to use this process to write (and rewrite) specifications for Java 1.7 as development on the new JML tool suite, OpenJML, continues.

[8] See `http://kindsoftware.com/trac/mobius/browser/src/mobius.esc/escjava/trunk/ESCTools/Escjava/test/jdktests` for details.

6 Conclusion

We have described a new process, the Formal CTD Process (FCTD), for determining whether formal specifications written for an existing software system are correct and useful. FCTD effectively uses the existing unit test suite for the software system as a behavioral specification and validates the formal specifications against the unit test suite by performing modular static verification. By doing so it ensures that the formal specifications capture enough of the system's behavior to pass the unit tests, demonstrating the utility of the specifications.

FCTD is best suited to testing library specifications using conformance tests, since conformance tests by definition describe exactly the required functionality of a library. We have described a method for applying FCTD to the standard Java class library using the Java Compatibility Kit and a version of its accompanying JavaTest infrastructure augmented with formal specifications for key classes. This will allow us to evaluate new specifications written for Java library classes against the library conformance tests used by all Java licensees, and thus to ensure that our library class specifications are of high quality.

While the concrete processes we have described here are specific to Java, JML, and their associated tools and tests, the general technique of using static verification of unit tests to validate specifications written for existing software systems is widely applicable. Our process can be easily adapted to any programming language and specification language for which both unit testing frameworks and static verification tools are available. We believe, therefore, that our process has the potential to significantly improve the quality of specifications written for existing software systems, and thereby also to significantly increase the utility of formal verification techniques that rely on such specifications.

There are several open research challenges and opportunities associated with this work. We speculate that unit tests generated by tools that are unaware of specifications (e.g., those that perform symbolic execution, shape analysis, etc.) may have some utility in terms of testing specifications, but we have not yet explored this avenue. We also speculate that the output of *specification inference* tools such as Daikon [6] and Houdini [7], which attempt to infer preconditions, postconditions and invariants from a body of code, may provide good starting points for the generation of correct and useful specifications. Finally, we believe it is possible to provide a more precise measurement of code and specification coverage than the "number of tests" ratio we currently use as a measure of utility.

References

1. Beust, C., Suleiman, H.: Next Generation Java Testing. Addison–Wesley Publishing Company (2007)
2. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., T.˜Leavens, G., M.˜Leino, K.R., Poll, E.: An Overview of {JML} Tools and Applications. International Journal on Software Tools for Technology Transfer (Feb 2005)
3. Chalin, P., Robby, et al.: JMLEclipse: An Eclipse-based JML specification and verification environment. http://jmleclipse.projects.cis.ksu.edu/ (2011)

4. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Magnusson, B. (ed.) Proceedings of the European Conference on Object-Oriented Programming (ECOOP) 2002. Lecture Notes in Computer Science, vol. 2374, pp. 231–255. Springer-Verlag (Jun 2002)
5. Cok, D.R., et al.: OpenJML. http://sourceforge.net/apps/trac/jmlspecs/wiki/OpenJml (2011)
6. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The {Daikon} System for Dynamic Detection of Likely Invariants. Science of Computer Programming 69(1-3), 35–45 (Dec 2007)
7. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: International Symposium of Formal Methods Europe (FME). Berlin, Germany (March 2001)
8. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for {J}ava. ACM SIGPLAN Notices 37(5), 234–245 (2002)
9. Free Software Foundation, Inc.: GNU Classpath. http://www.gnu.org/software/classpath/ (2011)
10. Gamma, E., Beck, K.: JUnit: A regression testing framework. http://www.junit.org/ (2011)
11. Hyland, R.: A Process for the Specification of Core JDK Classes. Master's thesis, University College Dublin (Apr 2010)
12. Janota, M., Grigore, R., Moskal, M.: Reachability analysis for annotated code. In: 6th International Workshop on the Specification and Verification of Component-based Systems (SAVCBS 2007). Dubrovnik, Croatia (Sep 2007)
13. Kiniry, J.R., Cochran, D., Tierney, P.: A verification-centric realization of e-voting. In: International Workshop on Electronic Voting Technologies (EVT) 2007. Boston, Massachusetts (2007)
14. Kiniry, J.R., Cok, D.R.: ESC/Java2: Uniting ESC/Java and JML. In: Proceedings of CASSIS 2004. Lecture Notes in Computer Science, vol. 3362. Springer-Verlag (Jan 2005)
15. Kiniry, J.R., Zimmerman, D.M.: Secret ninja formal methods. In: Proceedings of the Fifteenth International Symposium on Formal Methods (FM). Lecture Notes in Computer Science, vol. 5014 (2008)
16. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. In: Proceedings of FMCO 2002. Lecture Notes in Computer Science, vol. 2852, pp. 262–284. Springer-Verlag (2003)
17. Meyer, B.: Object-Oriented Software Construction. Prentice–Hall, Inc., second edn. (1988)
18. Oracle Corporation: JT Harness project. http://jtharness.java.net/ (2011)
19. Oracle Corporation: OpenJDK. http://openjdk.java.net/ (2011)
20. Oracle Corporation: OpenJDK community TCK license agreement. http://openjdk.java.net/legal/openjdk-tck-license.pdf (2011)
21. The Apache Software Foundation: Apache Harmony - Open Source Java SE. http://harmony.apache.org/ (2011)
22. The Eclipse Foundation: The Eclipse project. http://www.eclipse.org/ (2011)
23. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Computing Surveys 29(4), 366–427 (Dec 1997)
24. Zimmerman, D.M., Nagmoti, R.: JMLUnit: The next generation. In: International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010). Paris, France (Jun 2010)