Logical Testing

Hoare-style Specification Meets Executable Validation

Kathryn E. Gray and Alan Mycroft

University of Cambridge Computer Laboratory
{Kathryn.Gray,Alan.Mycroft}@cl.cam.ac.uk

Abstract. Software is often tested with unit tests, in which each procedure is executed in isolation, and its result compared with an expected value. Individual tests correspond to Hoare triples used in program logics, with the pre-conditions encoded into the procedure initializations and the post-conditions encoded as assertions. Unit tests for procedures that modify structures in-place or that may terminate unexpectedly require substantial programming effort to encode the postconditions, with the post-conditions themselves obscured by the test programming scaffolding. The correspondence between Hoare logic and test specifications suggests directly using logical specifications for tests. The resulting tests then serve the dual purpose of a formal specification for the procedure.

We show how logical test specifications can be embedded within Java and how the resulting test specification language is compiled into Java; this compilation automatically redirects mutations, as in software transactional memory, to support imperative procedures. We also insert monitors into the tested program for coverage analysis and error reporting.

1 Introduction

A unit test comprises a statement of the initial conditions, a statement or expression to evaluate, and a statement of concrete expectations for the result — in essence, a Hoare triple. The similarity of these components and their roles suggests that test specifications can draw from logical specifications in syntax and semantics. With similar specifications, tests serve as both runtime and logical validation; however, directly encoding tests into a standard programming language complicates the specification of some tests, making the connection to logical specifications needlessly complicated.

In discussing test specifications, we separate procedures into two broad classes: *generative* procedures that construct new data and *imperative* procedures that modify data structures and redirect bindings. As generative procedures do not modify any values, the associated test specifications closely resemble Hoare-style post-conditions that only involve the current values of variables. Post-conditions for imperative procedures in general need to refer to an initial value, commonly referred to as *old*, as well as the current, modified, value. Providing access to both values in previous executable tests obscures the relationship to

M. Chechik and M. Wirsing (Eds.): FASE 2009, LNCS 5503, pp. 186–200, 2009.

logical specifications and can cause crucial post-conditions to be omitted from test specifications.

Accessing the old value in an executable test requires a copy of the initial value that will not be modified. For unstructured data, such as integers, making the copy is simple. For structured data, such as an object with fields, copying the initial value is difficult and can lead to test specifications that incorrectly encode the post-conditions. The object-copy should copy each field, to identify sub-structure modifications, but heap-level equality is lost.

Given these problems in test specifications for imperative procedures, as well as additional problems caused by non-standard termination (i.e. exceptions), we bypass the programming language and use a specialized extension of Java expressions for test-specification. We extend Java with testing forms based on Java Modeling Language [11] (JML) guarantee specifications. Our compiler generates Java programs from test specifications that provide references to both unmodified and modified versions of objects using software transactional memory (STM) [12].

Snapshot Testing

In STM, modifications to a transactional variable affect the value read for some clients while others read the initial value. Modifications to the transaction are stored until a command commits them to the value, exposing them to all clients, unless a conflict causes the modifications to be aborted. We use these techniques to isolate modifications to our initial value instead of copying structural data.

Consider a method that imperatively inserts an item into a list with the post-condition that after insertion the list is a strict superset of the initial list. Copying the list requires making a copy of each element of the list, which may themselves be complex data structures. A copy can be memory intensive, may require duplication of externally-shared resources that should not be copied (such as network sockets), and prohibits pointer-equality comparisons. For some post-conditions, such a copy may invalidate the test.

Instead of copying values, we introduce the idea of *snapshot tests* that automatically preserve the original data structure while also supporting access to the modified version. In databases and file-system backups, access to older versions of the system are provided through snapshots, inspiring our name. Due to storage space involved, it is infeasible for these systems to perform a bit-for-bit copy and instead store a delta of the changes made. Similarly, our snapshots store the modifications made to the value during a test and we treat the original value as a transaction to record the modifications. During the test command, reads to the snapshot value use the transaction's record (a 'log'). The post-conditions can read either the log or the original value. Subsequently, we commit the changes.

Logical Specifications and Assertions

Hoare-style logical annotations have been incorporated into Java both as formal specifications (i.e. using JML), and assertions. While these specifications, as generalized descriptions of a method's behavior, complement verification

techniques, they do not fill all of the needs of a test. A test sets out all specific initial conditions for evaluation and concisely connects these to the specific outcomes; a formal specification does not necessarily provide sufficient information to evaluate a command.

While JML can be used for verification and static analysis, such as in tools like ESC/Java [7], it also comprises the syntax of tools that convert the specifications into runtime assertions. These tools have taken the first steps toward integrating formal specifications and execution-based validation. However, their formal specifications do not support all of the standard aspects of test development; nor do these implementations fully support accessing initial values after a mutation.

The Jass Java-extension [2] embeds a pre- and post-condition language into Java comments, using a subset of JML. These conditions are compiled into assertions within the method; however, a user must still provide specific method calls in a separate location to create tests. The separation between method call and post-conditions increases development efforts and reduces the clarity of a test. Jass specifications support access to both the initial and modified values bound to a variable, using an `Old` designation. However, the implementation for this feature uses the Java `clone` method, which duplicates top-level field bindings but does not necessarily recur into structures to clone the reachable data. Depending on the semantics of any comparisons, this clone can either not provide sufficient duplication or can break pointer-level comparisons. Our snapshot representation dynamically preserves both qualities, so that the test specification can freely describe the expected behavior.

Another similar system, the Cheon and Leavens [6] system, generates JUnit test specifications based on JML specifications but requires programmers to modify the generated test cases with specific parameters and sometimes results, also provides connections between formal specifications and runtime validation. However, like the Jass implementation, this system does not support testing imperative methods.

Existing hybrid systems either preclude the concrete executions of a test case or split test specification into multiple locations. Also they do not properly support the semantics of accessing the initial data after an imperative method. Our work builds on this background to provide a specification framework that matches standard test development and supports more accurate specification semantics.

Roadmap

Our test specification forms, presented in Section 2, allow test specifications to support both logically valid assertions with specific executions, combining the best of existing formal specifications and validating test environments. We demonstrate the ease of specification, compared with the industry-standard test specification library JUnit in Section 3. With our specialized test specification forms, we can provide additional information regarding test evaluation and failure than standard test engines, described in Section 4.1. Section 4.2 outlines compiling these test specifications into standard Java expressions, with a specific emphasis on compiling snapshots in Section 5.

2 Test Specifications

Hoare-logic based specifications, as typified by JML, annotate method definitions with requirements and guarantees. These specifications encode a generalized view of the method's behavior. In contrast, a typical test specification examines a concrete evaluation of a method call and encodes the specific behavior for the circumstance. Due to the different requirements, the test specifications must examine individual evaluations – typically a unit test examines one method call, so we present this circumstance.

We present our test specifications relative to a class modeling a board game

```
class Board {
  Board(String size, Piece initial) ...
  void slide(Piece p, char d) ...
}
abstract class Piece {  Posn left; }
```

A typical test first establishes a set of bindings – the precondition.

```
Piece p = new LShape();
Board b = new Board("small", p);
```

A test precondition comprises a set of variable declarations and data initializations, as above. A test *command* uses the variable declarations

```
b.slide(p, 's')
```

Following this imperative method call, the test specification confirms that necessary post-conditions are met; our extended Java expression language encodes these specifications in JML-style syntax, for example

```
modifies(b) && old(p).left.y  == p.left.y + 1
```

This *TestExpression* ensures that the `slide` method changed the internal board structure and updated the specified piece's left corner to a lower coordinate.

To deal with imperative test specifications, *TestExpressions* extend the Java expression form with three variants, see Figure 2[1]. Each *TestExpression* produces a boolean indicating the success or failure of the individual post-condition. The `old` designation allows the specification to access the snapshot of value bound to the specified variable; the `modifies` predicate determines whether the test command has performed any mutations involving the value bound to the specified variable; and the `modifiesOnly` predicate determines if the test command has performed a mutation involving only the specified fields of the specified variables, leaving other bindings unmodified.

Each *TestExpression* evaluates with respect to a particular tested command, in our example the `slide` method call. To identify the tested command, our expression test form combines the command with the post-conditions, e.g.

```
b.slide(p,'s') ensure modifies(b) && old(p).left.y  == p.left.y + 1;
```

[1] We accept any Java expression with nested *TestExpressions* as a *TestExpression*.

$$
\begin{array}{lll}
\textit{TestExpression} ::= & [\textit{TestExpression}\,/\,\textit{Expression}] \\
& \mid \textbf{old}(\textit{Variable}) \\
& \mid \textbf{modifies}(\textit{Variable}) \\
& \mid \textbf{modifiesOnly}(\textit{Variable.Name}[, \textit{Variable.Name}]*) \\
& \mid \textbf{throws}\ \textit{Name} \\
\textit{Expression} ::= & \cdots \\
& \mid \textit{Expression}\ \textbf{ensure}\ \textit{TestExpression}
\end{array}
$$

Fig. 1. Test Specification Forms[2]

The infix[3] `ensure` keyword is drawn from the JML keyword for method requirements, and identifies the scope of the post-conditions. The *TestExpressions* include the existing expressions, where *TestExpressions* can occur within the expressions, and each produce booleans. The `throws` form of *TestExpression* determines whether the tested command terminated with the named expression, while the others provide a means of describing modifications to data structures. Further post-conditions can be evaluated in conjunction with a tt throws test, effectively merging the `ensures` and `exsures` keywords from JML.

2.1 Language Additions

While the *TestExpressions* theoretically suffice to encode test specifications, for practical purposes (including error reporting) we further extend the Java syntax to support test development. We also provide test-specific top-level grouping constructs to simplify test specifications, discussed in greater detail in Section 3.2, as well as further *TestExpression* forms that provide more convenient specifications. Figure 2 presents all of our additions.

$$
\begin{array}{lll}
\textit{TopDef} ::= & \cdots \\
& \textbf{test}\ \textit{Variable}\ \{\ \textit{DefMember}\ \} \\
\textit{DefMember} & \cdots \\
& \textbf{testcase}\ \textit{Variable}\ \{\textit{MethodBody}\} \\
\textit{Expression} ::= & \cdots \\
& \mid \textit{Expression}\ \textbf{ensure}\ \textit{TestExpression}' \\
\textit{TestExpression}' ::= & [\textit{TestExpression}'\,/\,\textit{TestExpression}] \\
& \mid \textit{TestExpression} \\
& \mid \textbf{result} \\
& \mid \textit{TestExpression}' \mathrel{===} \textit{TestExpression}' \\
& \mid \textit{TestExpression}'\ \textbf{memberOf}\ \textit{Expression}
\end{array}
$$

Fig. 2. Test Specifications

The `result` binding, following the JML specifications, provides the result value of the test command on the right-hand side of the `ensure` expression. We

[2] The form $[a/b]$ denotes all the right-hand sides of productions for b, but with occurrences of b replaced by a.

[3] `ensure` follows the same precedence as the `?` operator in conditional expressions.

restrict binding local variables within a *TestExpression* for clarity and compilation concerns, therefore we must automatically provide a binding for this value. A *TestExpression* may only occur within a `test` definition. The final two forms perform common comparisons; a structural comparison of two values using `===`, and a comparison between the stated value and one of an array of values in `memberOf`. Programmers could write these specifications themselves, but it is convenient to incorporate such forms into the language.

3 Comparison with JUnit

Encoding tests as Hoare-style specifications rather than as stylized calls within a test suite leads to clearer test specifications as well as greater connections between executable validation techniques and formal verification. To explore the improvements for specifications, we compare our test specifications with test specifications written in the most prevalent testing package for Java, JUnit. These sample test specifications evaluate an extension of our board game example from Section 2.

3.1 Individual Tests

A JUnit test specification uses assertion methods, such as `assertEquals`, to validate test properties, often augmented with logging information to aid in test reports. These methods are provided from a class `Test`, that programmers extend. Programmers may add specialized assertion methods that combine the initial assertion methods.

Comparing Non-Structured Data. Comparing non-structured values, such as integers, in JUnit can use the standard assertion method. This method accepts two values and uses either numeric equality or the Java `equals`, which defaults to pointer equality, method to compare them.

```
assertEquals("board size", b.size(), 30);
```

The `Test` class provides an appropriate `assertEquals` method for all values.

Comparing values in *TestExpressions* uses `result` to access the test command's returned value.

```
b.size() ensure result == 30
```

Test specifications may require multiple checks on a given value, in which case the JUnit test must also store the resulting value in a variable, for example when a partly random result is expected:

```
int size = b.size();
assertTrue("board size", (size > 20 && size < 50));
```

versus our specification

```
b.size() ensure (result > 20 && result < 50)
```

These simple specifications are not significantly different in either implementation, due to their simplicity. As a language extension, failure reports between our system and JUnit differ, these differences are discussed in Section 4.1.

Checking Imperative Methods. With imperative methods, comparisons in JUnit test specifications require the programmer to either explicitly copy an existing value or manually compare individual fields to constant values. The following example uses both techniques

```
Piece p2 = new Square();
Board bOld = copy(b);
b.place(p2, 7, 9);
assertTrue(bOld.grid.subset(b.grid));
assertEquals(p2.left.x, 7);
assertEquals(b.numMoves, bOld.numMoves);
assertEquals(b.maxSide, bOld.maxSide);
...
```

This test specification correctly validates the performance of `place` provided a) the `copy` procedure properly copies all the field values so that the two boards do not share values, particularly the grid, b) the `copy` method preserves the necessary information for the other fields to satisfy the equality methods, and c) all of the fields are listed. As a program develops over time, the likelihood of the test correctly validating the method decreases as fields are added and implementations change.

Our specification uses a snapshot of `b` and `p2` to validate the post-conditions.

```
Piece p2 = new Square();
b.place(p2, 7,9) ensure (modifiesOnly(b.grid, p.left) &&
                         old(b).grid.subset(b.grid))
```

The `modifiesOnly` form checks that for the listed bindings, namely `p` and `b`, the referenced fields are modified without changes to any other fields in these objects. The `old` form allows the specification to access the initial grid value.

Checking for Exceptions. Testing methods using JUnit that may cause an exception requires either explicitly placing the method call in a `try` block or passing any exception along to the test engine. For simple tests and intricate interactions, this can be too large a burden for programmers to implement correctly.

The `ensure` expression removes the need to explicitly handle exceptions within the specification, with the `throws` test clause providing a per-call means to evaluate error handling implementations.

In JUnit, a test specification that anticipates an exception can omit a `try` block if restricted to one method call and exception per test case, as shown in the following example

```
@Test(expected = IllegalMove.class) void place() throws IllegalMove {
  Board b = new Board("small");
  b.place(new Square(), -10, 0)
}
```

The annotation at the method definition, contained within the test class, indicates that an exception must halt the execution of this test method and any

other behavior is an error. Grouping this test with previous test specifications manipulating the board is problematic; additionally, testing that a method has performed any mutations prior to raising an exception cannot be handled in this style.

To show the potential problems caused in using JUnit, the following protocol[4] requires that multiple exceptions be tested within one method, to ensure that proper side-effects occur during exception handling.

```
Piece p1 = ..., p2 = ...; Posn c = ...;
b.place(p1,c);
try {
  b.place(p2,c);
} catch( ContestedPosition e) {
  try {
    b.location(p1);
  } catch( UnplacedPiece e) {
    return;
  }
  fail("UnplacedPiece not thrown");
}
fail("ContestedPosition not thrown");
```

The second call to `place` attempts to overlay one piece on another, which causes an exception and changes the state in `p1` and `b`. The call to `location` should now fail due to the state changes caused in modifying the board due to the encountered error. Correctly developing such nested `try` blocks can lead to errors with omitted `return` statements or misplaced calls.

An equivalent test specification using our `ensure` expressions follows

```
Piece p1= ..., p2= ...; Posn c = ...;
b.place(p1,c);
(b.place(p2,c) ensure throws ContestedPosition &&
 b.location(p1) ensure throws UnplacedPiece)
```

This eliminates the need for nested `try` statements and the second test specification clearly relies on the first.

Comparing Objects. For object comparisons, JUnit's comparison method uses the inherited `equals` method, which does not always perform the necessary structural comparison, so the programmer must develop an independent comparison. These cases are most problematic with arrays and with classes without source, where a structural comparison is necessary but unavailable. In JUnit, such a comparison follows

```
boolean comp(Piece[] a1, Piece[] a2) {
  boolean res = a1.length == a2.length;
  if (res)
    for(int i; i< a1.length; i++)
```

[4] Inspired by a test seen in a open-source text-editing project.

```
        res &= a1[i].equals(a2[i]);
    return res; }

assertTrue(comp(b.getPieces(3), new Piece[]{new LShape(), ...}));
```

The `comp` method correctly assess whether the two `Piece`arrays are equivalent. Depending on the comparison method for a `Piece`, different array comparison methods may be required. The `===` comparison simplifies these test specifications by providing a structural comparison for any two values.

```
b.getPieces(3) ensure result === new Piece[]{new LShape(), ...}
```

Comparisons of objects with private fields highlights an additional benefit offered with `===`; writing such a comparison can be problematic for a programmer, relying on reflection and security accesses. We leverage compile-time information to provide a structural comparison in all circumstances.

3.2 Test Organization

Test specifications typically occur in a separate Java package from the primary implementation, following the JUnit style. Individual classes are tested by an extension of the JUnit `Test` class that contains a set of methods which each test the behavior of a particular method in the implementation class. Thus in our example, a test-specification `Board` class would extend the `Test` class and primarily test the operations within the `Board` implementation.

We mirror this organization within our testing specifications, to accommodate the expectations of test developers. Instead of deriving a particular class, however, we provide a third top-level form `test` that serves as the grouping mechanism for testing class implementations. The additional form allows us to restrict the placement of the `ensure` keyword, so that standard Java programs are unaffected.

The methods within a JUnit test class are annotated with an `@Test` attribute, signaling that these methods test a particular facet of the implementation. These methods must take no parameters and return no values (conditions which are checked via reflection at runtime).

We again mirror this organization, but use a specialized form that omits the possibility of dynamic signature errors. Test methods use a `testcase` modifier, a la `abstract`, and cannot specify attributes. A `testcase` may only appear within a `test`.

These additional forms provide static checks of test organization while not requiring test developers to modify their test organization strategies.

4 Implementation

The *TestExpression* and enclosing `test` forms can all be compiled to standard Java and during compilation can be automatically integrated with a report mechanism. This further simplifies test development and allows a program to be tested on a standard JVM implementation.

4.1 Integration with Test Reports

Each compiled *TestExpression* includes source information as well as annotations that provide further tools for evaluating test performance. Using the source information, a failing test report can identify the tested command and any values involved in the computation as well as the nature of the failure, for example if a method triggers a different exception than one declared in a `throws` clause. This information can aid the programmer in eliminating mistakes, in either their specification or the program. Using a system like JUnit, programmers manually annotate test specifications with this information, complicating the development.

By identifying a test specification, with the combination of the `test` and `testcase` designations and the `ensure` keyword, we can provide test-specialized coverage information when compilation is extended with coverage-tracking features. The compiler identifies the start of each test and inserts calls to select the coverage information collected during evaluation of the test command, or of the `testcase`. This information can be used to assess the scope of a test, aiding in debugging failed tests and determining the benefit of more test development. Other analyses, such as memory accounting, could also be automatically incorporated into test suites to improve program assessment.

4.2 Implementation

The `ensure` expression compiles into a Java method call that returns a boolean value, with the two expressions involved compiled into an anonymous inner class that evaluates the test command and the post-conditions. Figure 3 contains the compilation target for a general `ensure` expression, where the test command generates a value. Other than `result`, the variables within the `value` method are fresh to avoid name capture.

Targeting an anonymous class allows the tested command to expand into a sequence of statements that initialize the snapshots used within the assessment. Additionally, this expansion permits the test command to evaluate within a controlled framework where exceptions can be caught and evaluations can be run in a separate thread, controlled by the `addTest` method. Due to the inner class, local variable declarations must be treated as final within the *TestExpression*, so that these values can be passed into the inner class. This safety restriction does not restrict program validation, as mutating abstract variables within the post-condition does not provide validation on the correctness of the test command.

Each *TestExpression* compiles into an expression that evaluates the condition and provides information to the test report engine. The comparison expression expands into a method call to a comparison function within the test harness that inspects all of the fields of an object, whether private or accessible; the throws expression expands into an `instanceof` statement using the specified class and the `exn` binding. The expansion for the snapshot-specific expressions, are addressed in Section 5 with the explanation of how a snapshot is implemented.

A standard test specification

```
Expr1 ensure Expr2
```

compiles into

```
test.addTest( new TestClosure() {
              public boolean value() {
                 Object result = null;
                 boolean ans = false;
                 Throwable exn = null;
                 ⟨⟨ set snapshots as indicated by Expr2 ⟩⟩
                 try {
                    result = Expr1;
                 } catch (Throwable t) {
                     exn = t;
                 }
                 ans = Expr2 ⟨⟨ with compiled TestExpressions ⟩⟩ ;
                   ⟨⟨ unset snapshots ⟩⟩
                 return ans;
               }})
```

Fig. 3. Compilation Template for `ensure` with Value Generating Test

5 Snapshot Tests

Our snapshots mimic the effect of creating a copy in memory, but do not actually copy any values, which preserves the semantics of both pointer-level equality and structural comparisons. We divert mutation operations into a per-object log to preserve the original data-structure, and redirect accesses to the log during tests.

5.1 Taking a Snapshot in Restricted Java

Redirecting field accesses can be more easily explained in a restricted subset of Java than in the full language; therefore, we initially prohibit field accesses outside of specialized methods – fields may only be read in a field-specific `get` method and may only be modified in a `put` method. All methods, including the constructors, may only use these accessor methods when referring to a field binding. Both methods may read one additional field we add to each object – a boolean `stmOn`[5] field, with initial value of `false`.

While `stmOn` is `false`, field reads and writes proceed as normal. Using the `old` designation or either modification predicate changes the referenced object's `stmOn` to `true` prior to evaluating the test expression. While `stmOn` is true, field reads and writes pass through a log and do not modify the binding.

We represent the modification log with a hash-table, where the field name (combined with the class) is the key. On calls to the `put` method, the hash-table entry is updated with the provided value for the field binding. On calls to the `get`

[5] The `stmOn` field's actual name is generated to avoid collisions.

method, the field value is read when no entry exists in the hash-table, otherwise the hash-table value is used.

Before returning from the test call, all snapshots are reverted to normal by changing the `stmOn` to `false` and the modifications contained in the hash-table are committed to the respective field values.

This solution correctly redirects field accesses for object snapshots when all fields refer to unstructured values (i.e. integers, characters, or booleans). However, when a field refers to a structured value and the test command modifies the structure of this value (i.e. a modification of the form `p.left.y = 5`, where `p` is from our previous examples), the modification has not passed through a log as the `left` field is not a snapshot.

We must prohibit modifications on a snapshot from affecting the value until after the test, therefore we propagate the modification to `stmOn` on each initial read of a field value of a snapshot. On the first access of a field binding within a test, the value referred to by the field becomes a snapshot and modifications cannot affect the stored value. Further on each call to `put`, if the provided value is not yet a snapshot, `put` first sets the `stmOn` before storing the value.

While this Java subset is too restrictive for standard Java programs, a translation from any Java program into this subset is straightforward – the primary concern is to select a fresh name for the field methods and to preserve the shadowing of inherited fields.

5.2 Taking a Snapshot with Reduced Costs

Using a method call to access field values is not uncommon in Java programs and can typically be in-lined by an optimizing compiler, removing dispatch overhead. However, with the addition of the `stmOn` parameter, an optimizer may not be able to identify that field accesses can be safely in-lined. Thus the compilation strategy for taking a snapshot could negatively impact performance of all programs – this should be avoided.

For each field, compilation generates the accessor methods described above but only includes the implementation for the case where `stmOn` is false. This permits an optimization pass to remove the indirection for all field accesses. We incorporate the snapshot log by extending each class with a test-aware version. These classes each override the accessor methods of the original class with the body contents described in Section 5.1. The next step is in replacing instances of the original class with instances of the test-aware class in snapshot-contexts.

If we replace all constructor calls in the test program with their test-aware counterparts, than any objects initialized within the test specification can be preserved within a snapshot. However, this does not affect constructor calls not made within the test but made externally, such as a constructor that itself calls other constructors to initialize internal state. Therefore, without whole-program modifications, we cannot redirect the constructor calls and thus cannot turn every object into a snapshot-variant using this strategy.

A snapshot reflects the state of a value, without regard for the origin of this value. Therefore our snapshot-aware class extensions embed an instance of their parent class; each field access method dispatches to the embedded class instance and other methods defer to the super class. During a field access, the snapshot implementation defers to the test-aware field implementation, while protecting the initial value, while dynamic method dispatch selects the correct implementation to test.

For each value requiring a snapshot, we modify the binding to refer to the test-aware extension of the class with the initial value embedded within. Due to dynamic method dispatch, we cannot statically determine the test-aware class to instantiate. We use reflective techniques to identify the class and create the instance dynamically.

Uses of `old` within the post-conditions translate into an access of the embedded value. The `modifies` and `modifiesOnly` predicates translate into method calls which examine the hash-table of the specified snapshot. By examining the log, we differentiate unmodified values from a modification that has restored the original value before termination. Before returning, we record the modifications found in each hash-table into the embedded value to commit the changes.

5.3 Supporting Binary Libraries

The previous translation converts source files, ignoring (JVM) binary compilations. We advocate compiling from source where possible, to provide source-level information in test reports and analyses; however, this cannot always occur when using external (pre-compiled) libraries. External libraries must be made test-aware through byte-code transformations to redirect field accesses. We must rely on aspect-oriented style rewriting to modify the `getfield` and `putfield` byte-code instructions into appropriate method calls; however, this modification has not yet been implemented.

Using an aspect-oriented 'advice' to generate test-aware byte-code highlights the similarity between our transformations, which inject transactions into existing Java applications as well as test-report monitors, and generalized aspect-oriented programs that inject additional functionality into existing programs.

5.4 Problems with Snapshots

Due to the indirections on field accesses and the additional memory consumption, using snapshots has an impact on the performance of tested methods. While the run-time impact of snapshots has not been measured, we believe that any run-time overhead should be minimized to avoid increasing the cost of test suite evaluation. In future versions of our compiler, we intend to use a combination of ownership types and liveness analysis to determine whether each field in a class requires a snapshot to accurately validate a method's performance. When no snapshot is necessary, we will either use a copy or ignore the variable.

6 Related Work

In addition to the existing work joining executable validation with formal specification discussed in Section 1, much effort has been put into improving the organization and execution of tests. An early effort in this regards is the SUnit [3] system for Smalltalk, which supports organizing groups of related tests into classes with individual test methods containing multiple test cases evaluating a single implementation method. Individual evaluations use assertion methods, that may compare or assess values. Libraries following the SUnit philosophy now exist to support similar organizations for most programming languages, including JUnit [4] for Java, SchemeUnit for Scheme [13], and LIFT for Lisp [10].

JUnit provides a library of assertion methods, derivable classes, and an integrated report interface. New tests extend a class from the JUnit library and contain test methods, indicated either with a `@Test` attribute[9] or by appending `test` to the name of the method. Programmers use `String`-valued fields and parameters to document test cases. Both the SchemeUnit and LIFT systems use macros to extend the language with specific test forms, improving error reporting. Both provide language forms for test specifications, but encodings for imperative procedure remain difficult.

More modern efforts, such as TestNG [5], attempt to refine the test organization strategies of JUnit to support flexible test evaluation at the level of test methods but do not provide additional support for test specifications within the language as we do.

As with the macro-based libraries, the jMock [8] test engine embeds a test specification language into Java – via strings. These test specifications do not provide support for imperative post-conditions and the form of embedding can increase the difficulty of reading the test suites, as the programmer must distinguish between evaluated strings and documentary strings.

The Fortress programming language [1] contains built-in forms for declaring test procedures. An individual test case is marked using a `test` modifier, and a library provides functions to report and terminate failing tests. This language-based support does not extend to representing the test specifications, only benefiting the test organization and execution.

7 Conclusions and Further Work

We noted that existing unit test specifications correspond to restricted Hoare-logic formulae. We then observed that a more relaxed set of post-conditions, including `old(X)` (value of `X` during the precondition), `throws` (holds when an exception is thrown) and the like greatly simplify the code required for tests (using JUnit as a base for comparison), especially for imperative methods, which modify structures in-place. The extended forms of expressions *TestExpression* used in post-conditions can be implemented using a variant of Software Transactional Memory (STM).

While the mapping of our unit test constructs can be seen as a simple extended Java-to-Java translation, we show that this translation can also be done at the JVM level for pre-compiled libraries.

The system described is implemented in two systems, one with an alternate syntax and restricted functionality, which can both be downloaded from `www.professorj.org/testing`.

More expressive Hoare-logic formulas could also correspond to test specifications (for example, universal quantification might reasonably be treated as random testing) to further enhance test development. We leave this to future work.

Acknowledgements. We thank Matthias Felleisen for helpful conversations as we began our explorations of test development. This work was supported by (UK) EPSRC grant GR/F033060 "Linguistic Support for Test Development".

References

1. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr., G.L., Tobin-Hochstadt, S.: The Fortress language specification. Technical report, Sun (2007)
2. Bartezko, D., Fischer, C., Moller, M., Wehrheim, H.: Jass – Java with assertions. In: Workshop on Runtime Verification (2001)
3. Beck, K.: Simple smalltalk testing with patterns. The Smalltalk Report (1994)
4. Beck, K., Gamma, E.: Test-infected: programmers love writing tests. Java Report (1998)
5. Beust, C., Suleiman, H.: Next Generation Java Testing: TestNG and Advanced Concepts. Addison-Wesley, Reading (2007)
6. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, p. 231. Springer, Heidelberg (2002)
7. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proc. PLDI (2002)
8. Freeman, S., Pryce, N.: Evolving an embedded domain-specific language in Java. In: Proc. OOPSLA (2006)
9. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley, Reading (2005)
10. King, G.: LIFT — the lisp framework for testing. Technical Report 01-25, U. Mass. CS (2001)
11. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A Notation for Detailed Design, ch. 12. Kluwer, Dordrecht (1999)
12. Shavit, N., Touitou, D.: Software transactional memory. In: Principles of Distributed Computing (1995)
13. Welsh, N., Solsona, F., Glover, I.: SchemeUnit and SchemeQL: Two little languages. In: Proc. Scheme Workshop (2002)