*Technical Report*

UCAM-CL-TR-776
ISSN 1476-2986

Number 776

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

# System tests from unit tests

Kathryn E. Gray, Alan Mycroft

March 2010

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

*http://www.cl.cam.ac.uk/*

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

*http://www.cl.cam.ac.uk/techreports/*

ISSN 1476-2986

**Abstract**

Large programs have bugs; software engineering practices reduce the number of bugs in deployed systems by relying on a combination of unit tests, to filter out bugs in individual procedures, and system tests, to identify bugs in an integrated system.

Our previous work showed how Floyd-Hoare triples, $\{P\}C\{Q\}$, could also be seen as unit tests, i.e. formed a link between *verification* and test-based *validation*. A transactional-style implementation allows test post-conditions to refer to values of data structures both before and after test execution. Here we argue that this style of specifications, with a transactional implementation, provide a novel source of system tests.

Given a set of unit tests for a system, we can run programs in *test mode* on real data. Based on an analysis of the unit tests, we intersperse the program's execution with the pre- and post-conditions from the test suite to expose bugs or incompletenesses in either the program or the test suite itself. We use the results of these tests, as well as branch-trace coverage information, to identify and report anomalies in the running program.

# 1 Introduction: The continuing trouble with tests

Tests find bugs; developing and executing tests improves the robustness of real programs. But test specifications themselves are typically programs that may contain errors and omissions. Uncovering these bugs, and any program errors that are obscured, can be considerably difficult.

This effect is compounded when multiple programmers combine their isolated subsystems to create a product. Each developer may provide a set of *unit tests*, which lay out the necessary initial conditions and correctness conditions for an individual method. But if the integrated program calls a method without respecting the expected initial conditions, the resulting error may occur within a third method with no clear connection to the misused method. In principle *system tests* or *integration tests* can catch these errors, but many such errors evade these tests into releases. This paper demonstrates that unit-test specifications can provide an additional source of system tests and thereby increase the reliability of deployed systems.

Unit tests alone cannot expose all errors in a system, particularly ones arising from interactions between program *units* developed by different individuals. Such errors may only arise during full system executions – with shared data, shared program control, non-emblematic data usage, and undocumented requirements. For a minimal example, consider a priority-queue implementation using a sorted-list representation and the two methods `insert` and `max`. The `max` implementation returns the head of the list $l$, with pre-condition *ordered(l)* and a code-coverage-complete test-suite. The `insert` implementation is incorrect and is structurally the same as `cons`, but with pre- and post-conditions of *ordered(l)*. A code-coverage-complete test-suite contains two cases – `insert(3, [])` and `insert(4, [3])`. However, evaluating the program `max(insert(4, insert(5, [])))` incorrectly returns 4. At first glance, `max` appears to be at fault. However, the error is within `insert`, hidden by its inadequate test-suite. In this minimal example, a simple code review can quickly uncover the problem, but similar errors arise in real-world implementations that are not so amenable to review.

With increased support in the language and runtime system for test specifications, we can identify situations like this by assessing correctness during system-wide execution interspersed with evaluation of test conditions executed under isolation so that side-effects during test-condition evaluation are suppressed from mainline code execution. We draw on ideas from logic and formal specification to create unit tests (which resemble Floyd-Hoare triples with pre-conditions, a command, and post-conditions) and system tests. When proving correctness for a set of related procedures using logical specifications, the pre- and post-conditions propagate correctness information to call sites within other procedures. Similarly, a program running in *test mode* can use unit tests to ensure that the correctness conditions hold during whole program execution. Test mode, a specialized program execution mode, uses the pre- and post-conditions from the unit-test evaluation, along with collected execution data, to identify faults in both the set of tests for the program and in the program.

In our previous work [11], we introduced a domain-specific language for unit-test specifications in Java—TestJava—that relied on compile-time intervention to create a transactional execution for assessing method-level correctness. Transactions in our previous system caught modifications so that test specifications for imperative methods could

consider data functionally. We build on this technique to show that combining unit tests with transactional executions can be used to automatically collect test information during system integration.

Due to the possibility of test specification errors, and the exploratory nature of tests during test-driven development, not all tests executed during test mode will have clear successes or failures. Instead we supply error reports which indicate the possibilities of errors, such as a circumstance where the pre-conditions from a particular unit-test specification are satisfied but the execution-path data for the unit test and system execution vary—indicating that the test suite has not fully tested this method. We do not terminate execution for test failures. With our error reports, programmers can explore modifications to test conditions in the whole program without forcibly halting a program—which may produce the correct results despite failing one or more unit-test specifications.

We introduce test mode via an operational semantics in Sections 4 and 5. This semantics relies on a formalization of our previous unit-test support, presented in Section 3, which we augment with formal pre-conditions introduced in Section 2. Both unit testing and test-mode execution with transactions intervene in the runtime operations of a tested program, we show that these relations are weakly bisimilar to reductions of standard Java in Section 8.1 and so do not impact program correctness. Section 9 describes implementation techniques to support test-mode execution.

# 2 Unit tests with formal conditions: TestJava

Unit tests that follow the SUnit[1] philosophy [3], embed pre- and post-conditions into a standard programming language. In a unit-test specification, the correctness of one method is assessed under a precise initial condition. The pre-conditions (which includes method parameters) for a unit test may be partially implicit. As an example, a method `direction` may return "left" for inputs less than 0, "right" for inputs greater than 0, and "straight" for 0. A set of unit tests for `direction` may only test three concrete values (e.g. -1, 0, and 1) without stating the general ranges. Such a *c*oncrete test tends to reduce the benefit of test specifications as documentation, and in integration and regression testing.

We provide an extension to our unit-test specification language, TestJava [11], that identifies pre- and post-conditions with logic-based syntax and supports generalized parameters. As with our previous language for post-conditions, we build on the syntax of the Java Modeling Language [14] and the ideas of Hoare-triples for formal test specifications. With formal pre-conditions, test-mode execution can use unit tests to assess the correctness of executing methods.

TestJava provides constructs for organizing unit tests and tests for an entire class, as well as individual test specifications. We present our syntax, reduced for a formal model, in Fig. 1. An individual *unit-test specification* follows the form of a Hoare triple: `(pre) requiredBy call() ensure (post)`. The **old** and **modifies** conditions can only occur in post-conditions. The **requirement** bindings use $=$ and $<$ in our model.

[1] These comprise most unit tests.

$defT$ ::= **test** $cid_1$ { $testcases$ }

$testcase$ ::=
| **testcase** $mid$ { $stmtTBlock$ }

$stmtTBlock$ ::=
| $exprT$;
| $declT$ $stmtTBlock$

$opT$ ::= = | <

$declT$ ::= $type$ $id$ = $exprT$;
| **requirement** $type$ $id$ $opT$ $exprT$;

$exprT$ ::= **new** $cid$
| $exprT.fid$
| $exprT.mid(exprT_1 .. exprT_n)$
| **instanceof**($exprT$ $cid$)
| $exprT_1$ $op$ $exprT_2$
| ($exprT_1$) **requiredBy** $expr$ **ensure** ($exprT_2$)
| **old**($id$)
| **modifies**($id.fid$)
| **this**
| $id$
| $value$

Figure 1: TestJava (model) grammar

A single test for the `direction` method outlined above is

```
requirement Guide g = new Guide();
requirement int d = new Random().get();
(d<0) requiredBy g.direction(d) ensure (result.equals("left"))
```

The `result` variable binds the value generated by `direction()` and is always available in the post-condition. When a *declT* binding uses the `requirement` attribute, unit-testing reduction annotates the bound value with a data structure to collect runtime information. Our model requires variables used within a unit-test specification to be specified with the `requirement` attribute, a full system could use static fields as well. This annotation also provides a means for identifying the reachability of necessary values that are not observably reachable through a method's parameters (including `this`).

If we extend our test example to include external parameters and state, we may add set of maps to the guide and a current position modified by the method. Thus our unit-test specification is now

```
requirement Map ukmap = new BritishRoadMap();
requirement Guide g = new Guide();
requirement int d = new Random().get();
...
(ukmap.hasPosition(g.pos) && d<0) requiredBy
   g.direction(d)
   ensure (result.equals("left") && ukmap.leftOf(old(g).pos, g.pos))
```

As readers, we assume that the `Map` object `ukmap` is used within the `direction()` method but the test specification does not indicate how the method accesses `ukmap` since the object is neither a parameter nor referenced by a field in `g`. Section 4 addresses how our system identifies an appropriate value for test-mode validation of such a pre-condition. The provided `old` predicate, introduced in our prior work, refers to the `Guide` object as it was before `direction()`.

In creating unit-test specifications with generalized correctness conditions, `requirement` bindings should be able to represent inequalities, ranges, excluded sequences, and sets of predicates that define a parameter's or field's bounds. However, for conciseness, this paper only considers a single inequality binding (<). Although a set of predicates could be

used to specify object requirements, we presently only model specifications that bind an object using =. For unit-test reduction, inequality specifications generate random values, within the bounds, which could be used to validate a single test specification numerous times. Despite the opportunity for developing general conditions, we assume that many developers will continue to begin with concrete test specifications. Section 6 presents our strategy for automatically generating general conditions for use in test mode, to provide more accurate test results for debugging and suggest robust test suites for future validation.

Note that unit-test specifications can be embedded within the pre- and post-conditions of another unit-test specification. Building on our example, if the `Map` implementation requires access to an external resource, such as a network socket, method calls in `Map` may rely on having acquired the socket, and for correct behavior must close it when finished.

```
((!ukmap.connected) requiredBy ukmap.open() ensure (ukmap.connected)
  && P ) requiredBy ukmap.findPath() ensure ( Q &&
 (ukmap.connected) requiredBy ukmap.close() ensure (!ukmap.connected))
```

The unit-test specification on the first line states a pre-condition for `findPath()` as well as opening a connection for the unit-test evaluation. Similarly, the unit-test specification on the third line provides a post-condition while also closing the connection. The test represents a set of implications similar to the logical implication that $(pre_{open} \Rightarrow_{open} post_{open}) \Rightarrow_{path} (pre_{close} \Rightarrow_{close} post_{close})$ In test mode, such specifications provide temporal correctness specifications, discussed further in Section 7.

# 3 Validation with unit tests

For unit-test validation, each unit test reduces in isolation with respect to other specifications for the same or other methods. The actions of one method call should not impact the actions of a second call where the two are not explicitly related. We achieve this isolation, in the presence of mutation, by using transactional object representations. Previously [11], we explained our representation, *snapshots*, with a compilation technique. We now provide an operational semantics for our unit-test validation, augmented with support for the pre-conditions introduced in Section 2, which allows us to demonstrate that unit-test reduction and test-mode reduction do not alter the meaning of the tested program.

The operational semantics for a subset of Java, including statements and mutation, resembles existing Java semantics [6, 9, 12]. We use an evaluation-context small-step semantics with the relation $\langle E[expr], store\rangle \rightarrow \langle E[expr'], store'\rangle$, which requires compilation to resolve super calls to class names and replace field accesses with accessor method calls.[2] Figure 2 presents our source-level syntax, where test specification syntax, *defTs*, is in Fig. 1. The *src* annotations in methods and branching *stmt* forms model the source location of the code, i.e. line number and file, for use in execution traces; the *const* value represents integer constants. The evaluation context grammar is straightforward except for unit-test specifications, which do not evaluate subterms. The semantics resolves variables using value substitution.

[2]Our full semantics is available at `www.cl.cam.ac.uk/~keg29/testing`.

$prog$ ::= $defs\ defTs\ expr$ $\quad$ $def$ ::= **class** $cid_1$ **extends** $cid_2$ { $fields\ fieldAccs\ methods$ }

$method$ ::= $type\ mid(type_1\ id_1\ ..\ type_n\ id_n)$ { $src_1\ stmtBlock\ src_2$ }

$src$ ::= ⟨Representation of file-line source position.⟩

$stmtBlock$ ::=
- $stmt$
- $stmt\ stmtBlock$

$stmt$ ::= **return** $expr$;
- **if** $expr$ {$src_1\ stmtBlock_1$} **else** {$src_2\ stmtBlock_2$}
- **while** ($expr$) {$src_1\ stmtBlock$} $src_2$
- $expr$;

$expr$ ::= **new** $cid$
- $expr.fid$@**get**()
- $expr.fid$@**set**($expr'$)
- $expr.mid(expr_1\ ..\ expr_n)$
- **this**.$cid.mid(expr_1\ ..\ expr_n)$
- **instanceof**( $expr\ cid$)
- $expr_1\ op\ expr_2$
- **this**
- $id$
- $value$

$value$ ::= **true**
- **false**
- $const$
- **null**

$type$ ::=
- $cid$
- **bool**
- **int**

Figure 2: Modeled source program syntax. We represent repetition with a plural form of a term, i.e. *methods* represents 0 or more occurrences of *method*; we use *stmtBlock* instead of *stmts* to simplify related reduction rules.

$store$ ::= ($const$ , $heap$)

$heap$ ::= $\epsilon$
- $heap + loc \mapsto objectVal$

$objectVal$ ::= **null**
- $loc$
- ($cid$ ($fid_0\ value_0$) .. ($fid_n\ value_n$))
- (**snapshot** $cid\ fieldTableList\ locList\ loc$)
- (**snapshot** $cid\ fieldTableList\ locList\ loc$ **req**)

Figure 3: Runtime structures for test execution

Stores are represented by pairs $(z, h)$, seen in Fig. 3, where $h$ is a *heap* (mapping *loc*s to object values) and $z$ is a counter incremented by **new**. The counter in our store representation is used to tag heap locations arising from **new**, and allows us to show that two stores are isomorphic, see Def 8.1, in lieu of keeping a shared set of variable roots. During reduction, we extend our set of values with *objectVal* to represent locations, objects, and transactional objects (**snapshot**s). Unit test reduction uses three relations: $\underset{unit}{\longrightarrow}$ to reduce **testcase** implementations and construct test results; $\rightarrow_u$ to reduce the unit-tested *expr*; and $\rightarrow_c$ to reduce pre- and post-conditions while collecting test data.

In unit-test reduction local variables, instances of *declT*, with a `requirement` attribute embed the value in a **req** wrapper, shown in Fig. 4 rule reqOb.[3] Before reducing a unit-test specification, the **req** wrappings are removed from parameters, and the **req** tag is removed from heap-stored snapshots accessible through the parameter's fields using a $\hookrightarrow$ relation that builds a new store without the tag. Thus only **requirement** bindings that

[3]We specify our rules with Ott [18], for machine-readability and automatic formatting.

$$\frac{\boxed{defs \vdash \langle stmtTBlock, store, testReport\rangle \underset{unit}{\longrightarrow} \langle stmtTBlock', store', testReport'\rangle} \quad defs \vdash \langle E[exprT], store, \rangle \underset{unit}{\longrightarrow\!\!\!\!\rightarrow} \langle E[value], store', testSpecResults\rangle}{defs \vdash \langle \textbf{requirement}\ type\ id\ =\ exprT\ ;\ stmtTBlock, store, testReport\rangle \underset{unit}{\longrightarrow} \langle stmtTBlock[(\textbf{req}\ id\ (\textbf{eq}\ value)\ value)\ /\ id], store', testReport'\rangle}\ \textsc{req}$$

$$\frac{\begin{array}{l} defs \vdash \langle E[exprT], store, \rangle \underset{unit}{\longrightarrow\!\!\!\!\rightarrow} \langle E[loc], store', testSpecResults\rangle \\ store'(loc) : (cid\ fieldVals) \qquad loc' \notin store' \\ value = (\textbf{req}\ id\ (\textbf{eq}\ loc')\ loc') \\ store' = (const\ ,\ heap) \\ store'' = (const\ ,\ heap + loc') \mapsto (\ \textbf{snapshot}\ cid\ \epsilon\ \epsilon\ loc\ \textbf{req}\ ) \end{array}}{defs \vdash \langle \textbf{requirement}\ cid'\ id = exprT;\ stmtTBlock, store, testReport\rangle \underset{unit}{\longrightarrow} \langle stmtTBlock[value\ /\ id], store'', testReport\rangle}\ \textsc{reqOb}$$

Figure 4: Reduction relation for unit-test validation

are not reachable are tagged.

Evaluation of the unit test-specification expression, Fig. 5 conducts the primary work of unit testing. Reduction first evaluates the pre-condition using a $\rightarrow_c$ relation, which reduces *exprT* conditions while collecting runtime values to be used in compiling the condition for test-mode reduction. When the pre-condition reduces to **false**, the expression reduces to **false**. After executing the method call, the post-conditions evaluate with the same information collecting relation $\rightarrow_c$, with the method result substituted in for a *result* variable. Unit test expressions embedded in pre- or post-conditions evaluate using an equivalent $\rightarrow_c$ reduction rule, but must be evaluated with the rules which collect runtime information instead of the current relation. The **absObj** function collects additional data for test mode and is described in Section 6.1.1

When reducing the method call in TESTR, we use a reduction relation $\rightarrow_u$ that extends standard Java reduction to support snapshots. Figure 6 presents representative rules for accessing and modifying fields through a snapshot. A *fieldTable* is a mapping from *fid*s to values; $\mapsto$ represents a write, and the (*fid* : *value*) syntax represents a field which has been read by the method but not modified. To capture modifications that occur in objects referred to through fields, all accessed objects whether modified or simply read embedded into snapshots and stored in the *fieldTable*. As shown in SETSNAP writes do not impact the object location, and from GETSNAP3 reads use the *fieldTable* where applicable. When a write removes a location from the *fieldTable*, the *loc* is stored in the *locList* (see SETSNAP2) so that the snapshot can be properly committed.

# 4 Using unit tests as system integration tests

We model system execution as a sequence of definitions followed by a single method call, an instance of *prog* in Fig. 2. Test mode execution uses the same expression as system execution, while inserting test conditions for each method called. When method calls

$$\boxed{defs \vdash \langle E[exprT], store, testSpecResults\rangle \underset{unit}{\longrightarrow} \langle E[exprT'], store', testSpecResults'\rangle}$$

$$\frac{\begin{array}{l} defs \vdash \langle exprT,\ store\rangle \rightarrow_c \langle \mathbf{true},\ store_1,\ condition\rangle \\ defs \vdash \langle E[expr],\ store_1,\ \epsilon\rangle \twoheadrightarrow_u \langle E[(\mathbf{req}\ id\ (restrict)\ loc)],\ store_2,\ trace_1\rangle \\ store_2 \bullet (\mathbf{req}\ id\ (restrict)\ loc) \hookrightarrow store_3 \bullet loc' \bullet restrict' \\ \overline{defs \vdash \langle E[expr'_n],\ store_3,\ \epsilon\rangle \twoheadrightarrow_u \langle E[value'_n],\ store_4,\ trace_2\rangle}^{\,n} \\ \overline{store_4 \bullet value'_n \hookrightarrow store_5 \bullet value''_n \bullet restrict''_n}^{\,n} \\ defs \vdash \langle E[loc'.mid(\overline{value''_n}^{\,n})],\ store_5,\ \epsilon\rangle \twoheadrightarrow_u \langle E[value],\ store_6,\ trace\rangle \\ defs \vdash \langle exprT_2\,[value\,/\,\mathbf{result}],\ store_6\rangle \rightarrow_c \langle value_2,\ store_7,\ condition'\rangle \\ \mathbf{absObj}(condition,\ condition',\ loc' : restrict'\ \overline{value'_n : restrict''_n}^{\,n},\ store_7) = \\ \qquad restrict'''\ \overline{value'''_n}^{\,n},\ condition'',\ condition''' \end{array}}{\begin{array}{l} defs \vdash \langle E[(exprT)\ \mathbf{requiredBy}\ expr.mid(\overline{expr'_n}^{\,n})\ \mathbf{ensure}\ (exprT_2)], \\ \qquad store,\ testSpecResults\rangle \underset{unit}{\longrightarrow} \\ \qquad \langle E[value_2],\ \mathbf{commit}(store_7), \\ \qquad\ cid\ mid\ restrict'''\ (\overline{value'''_n}^{\,n})\ condition''\ condition'''\ trace\ value_4 \\ \qquad\ testSpecResults\rangle \end{array}}\ \textsc{testR}$$

Figure 5: Reduction of unit-test specification expressions

occur in sequence within the body of an outer method, the post-conditions of the first method feed into the establishment of the pre-conditions of the following method. If conditions fail, test reports provide the method and conditions responsible.

Unlike program verification, test-based validation does not expect to fully determine program correctness. Programmers may test a subset of relevant properties and still desire meaningful results. Due to exploratory development, a unit-test specification may use concrete values instead of general conditions, (e.g. the test may specify $f(10) = 15$ when $f(n) < n + 2$ holds.) Therefore for system-level test-based validation, we need more varied responses than success and failure. A test-mode execution results in a set of test results, as well as the result of evaluating the expression. The set of possible test results include the following seven outcomes:

**Success** All executed methods had matching pre-conditions, where the appropriate post-condition succeeds.

**Path Success** At least one executed method had no conclusively matching pre-condition, but followed a path consistent with a predicted pre-condition.

**Untested** At least one executed method had no test information.

**Incorrect call** At least one executed method had no matching pre-condition from path-exhaustive tests.

**Untested Path** At least one executed method had no matching pre-condition, and execution path was inconsistent with tested paths.

**Path Anomaly** At least one executed method had a matching pre-condition, but execution path was inconsistent with pre-condition path.

$$\boxed{defs \vdash \langle E[expr], store, trace\rangle \rightarrow_u \langle E[expr'], store', trace'\rangle}$$

$$\frac{\begin{array}{l} store(loc) : (\textbf{snapshot}\ cid\ fieldTable\ locList\ loc') \quad fid \notin fieldTable \\ defs \vdash \langle E[loc'.fid@\textbf{get}()],\ store\rangle \twoheadrightarrow \langle E[primVal],\ store\rangle \\ store' = store[loc.fid : primVal] \end{array}}{defs \vdash \langle E[loc.fid@\textbf{get}()],\ store,\ trace\rangle \rightarrow_u \langle E[primVal],\ store',\ trace\rangle}\ \textsc{getSnap}$$

$$\frac{\begin{array}{l} store(loc) : (\textbf{snapshot}\ cid\ fieldTable\ locList\ loc') \quad fid \notin fieldTable \\ defs \vdash \langle E[loc'.fid@\textbf{get}()],\ store\rangle \twoheadrightarrow \langle E[loc''],\ store\rangle \\ store(loc'') : (cid'\ fieldVals) \qquad loc''' \notin store \\ store' = store + loc''' \mapsto (\ \textbf{snapshot}\ cid'\ \epsilon\ \epsilon\ loc''\ ) \\ store'' = store'[loc.fid : loc'''] \end{array}}{defs \vdash \langle E[loc.fid@\textbf{get}()],\ store,\ trace\rangle \rightarrow_u \langle E[loc'''],\ store'',\ trace\rangle}\ \textsc{getSnap2}$$

$$\frac{store(loc) : (\textbf{snapshot}\ cid\ fieldTable\ locList\ loc') \quad fieldTable(fid) : value}{defs \vdash \langle E[loc.fid@\textbf{get}()],\ store,\ trace\rangle \rightarrow_u \langle E[value],\ store',\ trace\rangle}\ \textsc{getSnap3}$$

$$\frac{store(loc) : (\textbf{snapshot}\ cid\ fieldTable\ locList\ loc') \quad fid \notin fieldTable}{\begin{array}{c} defs \vdash \langle E[loc.fid@\textbf{set}(value)],\ store,\ trace\rangle \rightarrow_u \\ \langle E[value],\ store[loc.fid \hookleftarrow value],\ trace\rangle \end{array}}\ \textsc{setSnap}$$

$$\frac{\begin{array}{l} store(loc) : (\textbf{snapshot}\ cid\ fieldTable\ locList\ loc') \quad fieldTable(fid) : loc'' \\ store' = store[loc \hookleftarrow (\textbf{snapshot}\ cid\ fieldTable\ loc'' : locList\ loc') \end{array}}{\begin{array}{c} defs \vdash \langle E[loc.fid@\textbf{set}(value)],\ store,\ trace\rangle \rightarrow_u \\ \langle E[value],\ store'[loc.fid \hookleftarrow value],\ trace\rangle \end{array}}\ \textsc{setSnap2}$$

Figure 6: Access and mutation of snapshot-protected objects

**Failed Test** At least one executed method had a matching pre-condition, and corresponding post-condition failed.

Several conditions include path consistency as well as the results of evaluating test conditions. With the possibility of too-specific or unsatisfiable pre-conditions, we must either report limited test results or seek alternative sources for assessing acceptable behavior. We choose to provide additional test results by building potential test conditions based on runtime values and coverage data for the tested program. For this, we gather control-flow traces during unit evaluation for comparison during test mode. This is represented with the *trace* state variable of Fig. 6.

The *trace* data structure contains a sequence of the *src* labels stored within methods and branching statements. When a *src* location is visited and is already within *trace*, we produce a branched *trace* starting from *src* that contains sub traces visited subsequently. We collapse branch traces when all contained *src*s are identical; we merge a branch when control resumes in sync. Entry and exit to methods and while loops are used to identify re-synced execution.

The $\rightarrow_c$ relation, Fig. 7, collects runtime values for conditions in unit-test validation.

$$\boxed{defs \vdash \langle exprT, store\rangle \rightarrow_c \langle exprT', store', condition'\rangle}$$

$$\frac{\begin{array}{c} defs \vdash \langle exprT,\ store\rangle \rightarrow_c \langle loc,\ store',\ condition\rangle \\ defs \vdash \langle E[loc.fid],\ store'\rangle \rightarrow\!\!\!\rightarrow \langle E[value],\ store'\rangle \end{array}}{defs \vdash \langle exprT.fid,\ store\rangle \rightarrow_c \langle value,\ store',\ \mathbf{acc}\ condition\ fid\rangle}\ \textsc{checkAccess}$$

$$\frac{store(loc) : (\mathbf{snapshot}\ cid\ fieldTable\ locList\ loc')}{defs \vdash \langle \mathbf{old}((\mathbf{req}\ id\ (restrict)\ loc)),\ store\rangle \rightarrow_c \langle loc',\ store,\ \mathbf{old}(id)\rangle}\ \textsc{checkOld}$$

$$\frac{\begin{array}{c} store(loc) : (\mathbf{snapshot}\ cid\ fieldTable\ locList\ loc') \\ fieldTable[fid] : (fid \mapsto value) \end{array}}{\begin{array}{c} defs \vdash \langle \mathbf{modifies}((\mathbf{req}\ id\ (restrict)\ loc).fid),\ store\rangle \rightarrow_c \\ \langle \mathbf{true},\ store,\ \mathbf{modifies}(id.fid)\rangle \end{array}}\ \textsc{checkModifies}$$

$$\frac{\begin{array}{c} store(loc) : (\mathbf{snapshot}\ cid\ fieldTable\ locList\ loc') \\ fieldTable[fid] : (fid : value) \end{array}}{\begin{array}{c} defs \vdash \langle \mathbf{modifies}((\mathbf{req}\ id\ (restrict)\ loc).fid),\ store\rangle \rightarrow_c \\ \langle \mathbf{false},\ store,\ \mathbf{modifies}(id.fid)\rangle \end{array}}\ \textsc{checkModifiesF}$$

Figure 7: Building pre- and post-condition information for test-mode during unit testing

This *condition* structure contains variable references, concrete values, and abstract object representations. This data is used to compile the pre- and post-conditions for a method. Section 6 presents this compilation, while Section 5 presents the operation of test mode.

## 4.1 Producing test reports

Three of our seven test reports are easy to assess. When a single pre-condition and implied post-condition succeeds, the test report is success. Equally trivially, when a single pre-condition succeeds and implied post-condition fails, the test report is failure. If no *methodT* exists, then the test report is not tested. For multiple matched pre-conditions, we generate a test report for each outcome and do not consider multiple successful pre-conditions an error on its own.

More interesting results are generated when no pre-condition is conclusively satisfied. If all pre-conditions fail, then either we have an untested path or an illegal call. Which report applies relies on an annotation in *methodT* that specifies if the unit tests cover the control flow graph of the method. If all pre-conditions are inconclusive and cannot be resolved after execution, we consider the current trace. For any pre-condition with a matching test-mode trace, we report **path success** for this condition. When possible, we evaluate the implied post-conditions, combining path success with success.

If a pre-condition is successful, but the traces do not match, we report a path anomaly. A path anomaly indicates the pre-condition(s) did not identify the control-flow path and may be insufficient in establishing correctness.

$progC$ ::=
  | $defs\ snapShotDefs\ expr$

$snapShotDef$ ::=
  | **snap** $cid$ { $condResults\ methodTs$ }

$methodT$ ::=
  | $type\ mid(type'_0\ id_0\ ..\ type'_n\ id_n)$ { $testBranchs$ }

$testBranch$ ::=
  | $restrict\ preCondition\ condition\ trace$

Figure 8: Test mode syntax

# 5 Operational semantics of test mode

A test-mode reduction consists of reducing a single expression, using both class definitions and a set of **snap** definitions generated from unit test data, see Fig. 8. Each snap "class" models a class in *defs* that was validated during unit-test reduction; and each *methodT* in a **snap** contains the pre- and post-conditions for the corresponding tested method in **class**. A **snap** definition contains the method definitions for a **snapshot**, as a **class** definition contains the methods for an object. The *testRs* encode the results from test validation, with an individual *testR* representing the possible outcomes outlined in Section 4. A *test suite* for a particular *progC*, Fig. 8, is the set of *methodTs* used in a test-mode reduction of the method called in *progC*; this set can be used to determine regression test requirements when *defs* is modified. As with unit-test reduction, most rules of our $\rightarrow_T$ relation remove test-specific information from the current state and reduce under the Java $\rightarrow$ rules. Figure 9 presents two highly relevant reduction rules for test mode; the field access reductions strongly resemble those from unit-test reductions.

All objects in test mode are embedded within a snapshot immediately after allocation, see Fig. 9 rule NEWTM. The test-mode snapshot uses a nested transaction, represented as a sequential list of *fieldTable*s, as test conditions are evaluated before and after each method call. Consider an execution where a method, `direction()`, contains a call to another method `currLoc()`; if the post-condition of `currLoc()` requires that the method have no side-effect, then side-effects from `direction` should not be mixed with side-effects from `currLoc()`. Nested transactions allow each method call to observe mutations in isolation.

A method call in test mode evaluates pre-conditions for the method before invocation, and then evaluates the appropriate post-conditions to build a test report for the call, see Fig. 9 rule CALLTM. If the call raises a null exception (the only exception possible in our language), reduction immediately halts with the error. We present method call reduction in the context of the `direction()` method example in Section 2. We assume a programmer has written two additional unit-test specifications to cover the remaining possible results, and that the method parameter is 5. Each unit-test specification has been compiled into a *testBranch* specification within the `direction` *methodT* definition. A *testBranch* includes an implied pre-condition, *restrict*, for `this`, implied pre-conditions for each of the parameters, and a compiled version of the pre- and post-conditions; implied pre-conditions are compiled from the **requirement** bindings of Fig. 1. A $\Downarrow_s$ relation takes these implicit and explicit pre-conditions and returns any test branches that are **satisfied** or **inconclusive**. For `direction()`, the test branch whose pre-condition requires a parameter greater than 0 is returned.

Before dispatching to the method body for `direction`, for each parameter (and `this`)

$$\boxed{\mathit{defs}\ \mathit{snapShotDefs} \vdash \langle E[\mathit{expr}], \mathit{traces}, \mathit{store}, \mathit{testRs}\rangle \rightarrow_T \langle E[\mathit{expr}'], \mathit{traces}', \mathit{store}', \mathit{testRs}'\rangle}$$

$$\frac{\begin{array}{l}\textbf{class}\ \mathit{cid}\ \textbf{extends}\ \mathit{cid}_1\ \{\ \overline{\mathit{type}_n\ \mathit{fid}_n\ ;}^{\,n}\ \mathit{fieldAccs}\ \mathit{methods}\ \}\ \in\ \mathit{defs}\\ \mathit{loc} \notin \mathit{heap} \qquad \mathit{loc}' \notin \mathit{heap}\\ \mathit{heap}'\ =\ \mathit{heap} + \mathit{loc} \mapsto (\mathit{cid}\ \overline{(\mathit{fid}_n\ \mathit{defVal}_n)}^{\,n})\\ \mathit{heap}''\ =\ \mathit{heap}' + \mathit{loc}' \mapsto (\ \textbf{snapshot}\ \mathit{cid}\ \epsilon\ \epsilon\ \mathit{loc}\ )\end{array}}{\begin{array}{l}\mathit{defs}\ \mathit{snapShotDefs} \vdash \langle E[\textbf{new}\ \mathit{cid}], \mathit{traces}, \mathit{const}\ .\ \mathit{heap}, \mathit{testRs}\rangle \rightarrow_T\\ \qquad\qquad \langle E[\mathit{loc}'], \mathit{traces}, \mathit{const}+1\ .\ \mathit{heap}'', \mathit{testRs}\rangle\end{array}}\ \textsc{new}\mathrm{TM}$$

$$\frac{\begin{array}{l}\mathit{store}(\mathit{loc}) : (\textbf{snapshot}\ \mathit{cid}\ \mathit{fieldTableList}\ \mathit{locList}\ \mathit{loc}')\\ \textbf{snap}\ \mathit{cid}\ \{\cdots \mathit{type}\ \mathit{mid}(\overline{\mathit{type}'_n\ \mathit{id}_n}^{\,n})\ \{\ \mathit{testBranches}\ \}\cdots\} \in \mathit{snapShotDefs}\\ \textbf{class}\ \mathit{cid}\{\cdots \mathit{type}\ \mathit{mid}(\overline{\mathit{type}_n\ \mathit{id}_n}^{\,n})\ \{\mathit{src}\ \mathit{stmtBlock}\ \mathit{src}'\}\cdots\} \in \mathit{defs}\\ \mathit{defs}\ \mathit{store} \vdash\ \mathit{loc}\ \overline{\mathit{value}'_n}^{\,n} : \mathit{testBranches} \Downarrow_s\\ \qquad\qquad \overline{\mathit{restrict}_i\ (\overline{\mathit{value}''_n}^{\,n})_i\ \mathit{condition}_i\ \mathit{condition}'_i\ \mathit{trace}_i}^{\,i}\\ \mathit{store}_1 = \textbf{takeSnap}(\mathit{loc}\ \overline{\mathit{value}'_n}^{\,n},\ \mathit{store})\\ \mathit{traces}_1 = \textbf{traceCheck}(\mathit{src}\ \textbf{enter},\ \mathit{traces})\\ \mathit{defs}\ \mathit{snapShotDefs} \vdash \langle E_S[\mathit{stmtBlock}], \mathit{src}\ \overline{\mathit{trace}_i}^{\,i}\ \mathit{traces}_1, \mathit{store}_1, \mathit{testRs}\rangle \rightarrow\!\!\rightarrow_T\\ \qquad\qquad \langle \mathit{value}, \mathit{trace}\ \overline{\mathit{trace}'_i}^{\,i}\ \mathit{traces}_2, \mathit{store}_1, \mathit{testRs}'\rangle\\ \mathit{traces}_3\ =\ \textbf{traceCheck}\,(\ \mathit{src}'\ \textbf{exit}\ ,\ \mathit{traces}_2\ )\\ \overline{\mathit{defs}\ \mathit{store}_2 \vdash\ \mathit{loc}\ \overline{\mathit{value}'_n}^{\,n}\ :\ \mathit{condition}_i\ \mathit{condition}'_i\ \mathit{trace}'_i\ \Downarrow_r\ \mathit{testR}_i}^{\,i}\\ \mathit{store}_3\ =\ \textbf{commit}(\mathit{loc}\ \overline{\mathit{value}'_n}^{\,n},\ \mathit{store}_2\ )\end{array}}{\begin{array}{l}\mathit{defs}\ \mathit{snapShotDefs} \vdash \langle E[\mathit{loc}\ .\ \mathit{mid}\ (\ \overline{\mathit{value}'_n}^{\,n}\ )], \mathit{traces}, \mathit{store}, \mathit{testRs}\rangle \rightarrow_T\\ \qquad\qquad \langle E[\mathit{value}], \mathit{traces}_3, \mathit{store}_3, \mathit{testRs}'\ \overline{\mathit{testR}_i}^{\,i}\rangle\end{array}}\ \textsc{call}\mathrm{TM}$$

Figure 9: Test mode method call

$$\frac{\begin{array}{l} store(loc) : (\textbf{snapshot}\ cid\ fieldTable\ locList\ loc'\ \textbf{req}) \\ \textbf{reset}(fieldTable, loc', store) = const\ .\ heap \\ store' = const\ .\ heap[loc] \hookleftarrow (\textbf{snapshot}\ cid\ \epsilon\ loc') \\ trace' = trace\ :\ src\ (\ \textbf{req}\ id\ ) \\ \textbf{class}\ cid\ \{\cdots\ type\ mid\ (\ \overline{type_n\ id_n}^{\,n}\ )\ \{\ src\ stmtBlock\ src'\ \}\ \cdots\}\ \in\ defs \\ defs \vdash \langle E_S[stmtBlock[loc\ /\ \textbf{this}]\ \overline{[value'_n/id_n]}^{\,n}],\ store',\ trace'\rangle \rightarrow\!\!\!\!\rightarrow_u \\ \qquad \langle value,\ store'',\ trace'\rangle \end{array}}{\begin{array}{l} defs \vdash \langle E[(\ \textbf{req}\ id\ (\ restrict\ )\ loc\ )\ .\ mid\ (\ \overline{value'_n}^{\,n}\ )],\ store,\ trace\rangle \rightarrow_u \\ \qquad \langle E[value],\ store'',\ trace''\ :\ src'\rangle \end{array}}\ \textsc{callReq}$$

Figure 10: Capturing the first access of a non-reachable requirement

a new *fieldTable* is appended onto the *fieldTableList* using the **takeSnap** function. The *trace*(s) for the selected *testBranch*(s) are appended to the front of the list of *traces* under consideration, and the opening *src* tag is used to create a trace for the current invocation. The existing traces under consideration, placed into the state by previous pre-conditions, are compared with the current *src* for a match and tagged if they do not match; each traced statement performs a similar comparison and a second comparison is done when the method returns.

After reducing the method body to a value, the post-condition relation $\Downarrow_r$ considers the selected *testBranch*(s) to evaluate the post-conditions, using the value and the collected *trace*s. If `direction()` follows its specification, our result is `"right"`. However, if the *trace*(s) considered during reduction do not match the execution path taken, then the generated test result is a **path anomaly**. The next **commit** pushes writes made in the outermost transaction *fieldTable* to the next table, removes the table, and commits the snapshots of *locList* locations as well (where *locLists* is similarly a list of lists). Then the value is replaced into the context to continue program reduction.

In this discussion, we ignored our example test conditions' use of the `Map` object, `ukmap`. This value is necessary for the correct reduction of `direction()`, but test-mode execution has not been explicitly provided a value to consider. Since statically analyzing the program to identify the correct dynamically accessible value is too cost-prohibitive, we use the execution path data from unit testing to locate the correct value where possible. A *src* in the trace can be augmented with a flag during unit-test reduction that indicates the use of the needed value, here `ukmap`.

Recall that during unit-test reduction, evaluating a **requirement** added a **req** flag to a snapshot, Fig. 4. Any snapshot accessed during the test with such a flag is necessary for the test conditions but not accessed through a parameter. At the first such access in a unit-test reduction, the snapshot flag is removed and the trace is appended with a flag to mark the location, (**req** *id*). Figure 10 presents method call reduction on a required value.

Test mode reduction, using **traceCheck** overloaded to accept *loc*, collects the current location at a **req** flag for use in test conditions. Thus our initial evaluation of *testBranchs* for `direction()` cannot conclusively satisfy the given pre-condition as reduction is necessary to locate the object for the `hasPosition()` predicate. Inconclusive pre-conditions

are treated as successful except when generating test reports. After reduction, if the trace contains an `ukmap` bound object, the pre-condition can be re-evaluated in the post-condition relation.

Unit test reduction does not necessarily attach an appropriate flag to a trace, despite encountering an appropriate snapshot. The identification of the object must be conclusive for the pre-condition check to be meaningful. Therefore, if the path taken to the object is not clearly repeatable, the tag must not be appended. If the annotation is the unique difference between two sub-traces of a branch point, the annotation is omitted. When this occurs, any test condition using this binding can never be conclusively satisfied.

# 6 Compiling conditions

The test conditions used during test mode follow directly from the pre- and post-conditions specified for unit testing. However, when unit-test specifications contain tested methods that receive object parameters, contain conditions that refer to local variables, or refer to concrete values, compilation is necessary to transform the conditions so that they can evaluate in a different context.

During compilation we can also examine test conditions that contain non-general conditions and attempt to produce an equivalent condition which is more general. Our generated conditions may reflect program behavior better than the provided unit-test condition, as we can modify the specification based on evidence from test-mode reductions. This can ease the transition between a developmental, exploratory test suite, including concrete values and results, and a rigorous, generalized test suite used for mature validation and eventual regression testing.

## 6.1 Building test conditions for test mode

Some test conditions, such as those of the `Map` object `ukmap` in Section 2 rely on predicates over the same object that is the target of the tested method—i.e. they rely on the current object. Compiling such test conditions for test mode is trivial, replace the local variable reference for the object to the object embedded within each snapshot.

When local bindings are used in parameter lists and test-conditions, such as the parameter in our example, compilation redirects references to the appropriate parameter variable within the *methodT* body. Binding declarations for these parameters may use inequality bindings, tying the parameter to a random value. The $<$ specification in a binding requires that, as a pre-condition for the test-mode execution, the actual method parameters fit within this bound as well as any other pre-condition clauses.

Object references passed in as parameters (including the implicit `this` parameter) may contain field references which specify an inexact bound on the value. For example in the unit-test specification

```
requirement int timeOut < 100;  requirement Map ukmap = new Map();
... ukmap.timer = timeOut ...
(!ukmap.connected) requiredBy ukmap.open() ensure (ukmap.connected)
```

An additional pre-condition on the form of the `Map` object is necessary to capture the implicit requirement on the `timer` field. For such conditions, parameter checking on object

representations ensures that an object conforms to an abstract shape representation of the object encountered during unit testing.

### 6.1.1 Abstract object representation

An object abstraction records the runtime type of the object, as well as the declared type, and requirements for field values. The value bound to a `requirement` identifier is embedded within a special **req** object, see Fig. 4, with a notation of the restriction for the value. For our model, this restriction is either equality or $<$. When a unit-test specification expression is reduced, Fig. 5, the $\hookrightarrow$ relation which removes bf require wrappings and flags from the parameters also builds an initial abstraction of objects. Numeric fields in this abstract representation contain the relevant **req** wrapper.

For fields with object types, the abstract representation could contain another abstract object representation recursively building another full object representation. In many circumstances, this naive representation would collect too much information that we may not desire (such as concrete file descriptors or irrelevant large data structures). Instead the initial representation stores the **req** wrapping of the object location and we delay further abstraction for the **absObj** function that runs after the method call. The contextual information from the test conditions and the snapshot representations may suggest unimportant information within the object.

Any fields, or subfields, which were not accessed during the execution of the method are likely to be unimportant for the test and so do not require representation within the abstract object. Additionally fields directly probed within the pre-conditions can be ignored in the abstract representation, relying on the programmer's conditions for correctness. For an object field that is accessed but where its fields are not accessed, the abstraction stores the runtime type and checks for non-null values.

These abstractions may still produce unwieldy representations, particularly for large recursive data structures operated over by recursive methods. Therefore we impose a cutoff depth for field access of $n$ (we use $n = 3$ in our investigations). This cutoff is ignored when post-condition accesses or requirement specifications require a larger depth.

## 6.2 Building generalized test conditions

During program development test specifications often include concrete pre- and post-conditions, so that method calls during system execution fail to satisfy a pre-condition of any unit test. Such unit-test specifications aid in the design and implementation of methods. Considering a test for a method to add an element to a List

```
requirement List i = ...;
(i.writable()) requiredBy i.add(4) ensure (i.length == 1)
```

The typical reader of this specification probably concludes that `add` does not always set `length` to 1 despite the specification, but during test-mode reduction this post-condition will rarely succeed. Test mode should permit system-wide executions while implementations are in development to aid in incremental integration and test-based exploration. With generalized test conditions, test mode can provide more information in these circumstances than full failure (or illegal calls in the case of pre-conditions).

We must take care, however, that concrete condition which do encode full correctness are retained. To preserve these situations, when only one unit-test specification exists for a method, no assumed conditions are generated. With more unit-test specifications, we compare the specifications to construct a set of constraints which defines the allowable calling contexts for the method.

### 6.2.1 Building numeric pre-conditions

With numeric parameters, or field values, compilation attempts to generate a set of acceptable value ranges for the method based on the observed values and any provided pre- or post-conditions for use in an assumed condition. First we rewrite the set of test conditions into a canonical logical form to identify uses of the same identifier within the test conditions. Then we consider each numeric value in turn.

We refer to the variable under consideration as $r$. Each condition using $r$ contains the value for $r$ used during evaluation of that condition and each condition can examine the trace for references to $r$. We sort the values of $r$ from lowest to highest, including any uses in pre-conditions with inequality operations. The assumption is made that each test specification illustrates a branching point in the tested method.

If the smallest value for $r$ is below an inequality specification value, then the equality condition is retained. Otherwise, the bottom value is used as a potential upper-bound to the minimum represented number. Similarly if the largest value for $r$ is above an inequality specification, the equality is treated as a strict upper-bound without modifications, otherwise an inequality predicate is generated as a lower-bound to the maximum represented number. The other cases select the arithmetic mean. If $r$ is fixed for the tests, then no generalization occurs.

Traces for the different test specifications are also analyzed. When multiple test specifications have followed isomorphic traces, these specifications are considered together before applying the values for $r$ in the over all ranking. If $r$ varied during these executions, then a range is established for these test specifications that encompasses all numbers encountered.

All generalized conditions are tagged, with the original condition retained, for use during test-mode reduction. During test mode, the result of generated conditions is less conclusive than those written by the programmer. Since the condition may not fully reflect the correctness condition, success and failure are not clear. Modifications to the pre-condition based on observed behavior may be necessary when the condition passes but the generated trace does not match, or the post-condition fails. The current trace is compared to other test specification traces for the method; if a match is found, the new $r$ is used to refine the pre-condition and a path success occurs. When no conditions based on $r$ succeed, the traces are considered and $r$ is used to generalize the pre-condition for any matching trace.

When a generated condition is used within test mode, the test result includes the generalized pre-condition used. This can aid the programmer in inserting the condition into the test specification.

### 6.2.2 Generalizing object abstractions

Conditions for numeric fields within an object abstraction can be easily generalized following the same strategy outlined above. Further generalizations can be made to build acceptable sets of object types which are subtypes of the declared type for a field.

# 7 Nested test conditions and temporal properties

At the end of Section 2, we introduced nested unit-test specification expressions with an example that opened a socket in the pre-condition, used the socket during the method, then closed it afterwards. Such test specifications can be used to represent temporal safety and (some forms of) liveness properties. In unit-test reduction, the unit-test specification expressions in the pre- and post-conditions simply call the specified methods. Test mode reduction cannot call these methods; their side-effects may alter the semantics of the program.

Instead, any test specification expression within a pre- or post-condition clause is lifted from the enclosing expression during compilation. In its place we insert a call to check a stored result of evaluating the unit-test specification. For a pre-condition, this call uses a unique identifier in the appropriate *condResults* table of a *snapShotDef*, Fig. 8, to extract a result. For a post-condition, this call must delay resolution of the post-condition until (conservatively) program termination, when a similar unique identifier can be used.

In the context of our example, when the `open` method is called for an object, the result of the test condition is stored in the *condResults* table of the `Map` class. The table uses a unique identifier, generated during compilation for the test specification, and the *loc* of the current object to retrieve the condition. When running the method `findPath`, we use the location and identifier to retrieve the result from the table; if no entry is found the pre-condition fails. The post-condition produces an inconclusive result to be checked at program termination; at this point, if `close` for the object has not been evaluated with a successful result, the test fails. The location portion of the key is obtained through the same mechanisms we employ to access any object used in a test condition.

In a real implementation, we do not advocate storing all objects in the test results while executing the program. Updates could be encoded as call backs that resolve the post-condition as soon as the test is conducted.

Once a stored condition is evaluated, the result may never be recalculated. Thus if the post-conditions of a nested pre-condition are no longer valid when accessed, our semantics will not catch this program error. Further specification constraints can ensure that such conditions are rechecked if necessary to eliminate inaccurate results.

# 8 Correctness and Bounds

Our technique for deriving system test results relies on modifying the operational semantics of both unit-test execution and program evaluation. We now provide evidence that these modifications do not alter the result of program reduction. Additionally, our techniques store program traces and additional data from test executions; we show that this collection process is bounded in the complexity of the program and will not generate unlimited data.

## 8.1 Testing does not interfere with program correctness

Each of our three operational semantics evaluate programs in the presence of a store; therefore to consider whether our testing semantics respect the intention of the programmer, we must consider whether two stores contain the same values. Stores map generated locations to object values, where the `new` operation selects any location that is not presently used within the store. There is no guarantee that any of the semantic rules select the same location when running equivalent `new` operations, and we do not wish to impose such a restriction on the store representation as this would not model an implementation. Instead, we track the program-order of evaluation of `new` expressions and create a mapping of locations to this program order using tags.

**Definition 8.1.** Isomorphic store

$$store \cong store' \textbf{ iff } i = j \wedge heap \cong heap' \\ \textbf{where } store = i.heap, store' = j.heap'$$

and

$$heap \cong heap' \textbf{ iff} \\ \forall loc_i, loc_j.\ i = j \wedge (store(loc_i) : (cid\ fieldVals)) \Rightarrow \\ store'(loc_j) : (cid\ fieldVals') \wedge \\ (\exists loc'.\ (store'(loc') : (\textbf{snapshot}\ cid\ fieldTable\ locList\ loc_j) \wedge \\ \textbf{reset}(fieldTable, locList, loc_j, store')) \Rightarrow fieldVals \cong fieldVals')$$

and

$$\frac{n = m \wedge fieldVals \cong fieldVals'}{(fid\ loc_n)fieldVals \cong (fid\ loc_m)fieldVals'}$$

$$\frac{fieldVals \cong fieldVals'}{(fid\ primVal)fieldVals \cong (fid\ primVal)fieldVals'}$$

In each semantics, the store counter is increased only when extending the store with an additional location due to the evaluation of a `new` expression. And only these locations are tagged with the counter of their program order; no operational rule inspects the tag of a location so evaluation is not dependent on these tags. Store isomorphism relies on the **reset** function to update snapshot field values, which might contain cycles. The **reset** implementation updates the snapshot objects referred to by the *locList*before committing the updates of the current *fieldTable*. If a field wrote to a *loc* containing a **snapshot**, then **reset** extracts the snapshot's embedded $loc'$ and stores this reference in *fid* for the object. We rely on a $\forall$ to **reset** the referenced snapshot; thus avoiding cycles.

We can show that two programs that begin with isomorphic stores reduce to two new states which also have isomorphic stores. We use the following definitions to show that test-mode reduction preserves the meaning of a program.

**Definition 8.2.** reduce = $\langle prog, store \rangle$

**Definition 8.3.** testMode = $\langle progC, store_t, traces, testRs \rangle$

The test mode program, $progC$, and the original program, $prog$, are related by $\sim$ when the *defs* and *expr* of the unreduced $progC$ are textually equivalent to the corresponding portions of the unreduced $prog$.

**Definition 8.4.** Test mode Equivalence

$$
\begin{aligned}
&reduce \approx testMode \Rightarrow \\
&\quad \textbf{if } prog.expr = primVal \wedge progC.expr' = primVal \wedge store \cong store_t \\
&\quad \textbf{if } prog.expr = loc_i \wedge progC.expr' = loc_j \wedge i = j \wedge store \cong store_t \\
&\quad \textbf{if } reduce \rightarrow reduce' \wedge prog \sim progC \wedge store \cong store_t \\
&\qquad \textbf{then } \exists testMode'.\ testMode \rightarrow\!\!\!\!\rightarrow_T testMode' \wedge reduce' \approx testMode' \\
&\quad \textbf{if } testMode \rightarrow_T testMode' \wedge store \cong store_t \\
&\qquad \textbf{then } \exists reduce'.\ reduce \rightarrow\!\!\!\!\rightarrow reduce' \wedge reduce' \approx testMode'
\end{aligned}
$$

**Theorem 8.1.** *reduce* $\approx$ *testMode is a weak bi-simulation.*
*Proof sketch by co-induction on* $\rightarrow$ *and* $\rightarrow_T$

Most cases follow trivially. The test mode semantics rules for most expressions and statements erase *traces* and *testRs* from the current state and apply the $\rightarrow$ reduction. By inspection we know that no rules except `new` reduction increment or use the store counter. Both relations rely on the same evaluation context grammar, which we use to show the program order is preserved; thus a *loc* introduced by either **new** rule is tagged with $i$.

Interesting cases arise with field accesses and method calls.

For a field read through a snapshot at *loc*, we generate *store*$'$ by committing $store_t$. $store_t \cong store$, so *fid* from *loc*$'$ in *store*$'$ (where *loc*$'$ is the embedded location for *loc*) is isomorphic to field with $\rightarrow$ in *store*. By the definition of commit, the value read from $store_t$ is equal to that read from *store*$'$. Field modification follows similarly. Both rely on lemma for **commit** showing that committing retains only the last modification of a field and does not omit any modified fields.

Method calls in test mode use multiple relations before and after invoking the method body, the sole action of $\rightarrow$ method call reduction. Each relation requires a lemma proving that the resulting store is isomorphic to the original store and that primitive values are preserved. These follow a straightforward induction on their respective relations. Both method calls reduce in the same number of statement reductions from the body.

**Definition 8.5.** reduceE $= \langle E[expr], store \rangle$

**Definition 8.6.** unitTest $= \langle E[expr], store_u, trace \rangle$

**Definition 8.7.** Unit test Equivalence

$$
\begin{aligned}
&reduceE \approx_u unitTest \Rightarrow \\
&\quad \textbf{if } expr \in primVal \wedge expr' \in primVal \Rightarrow expr = expr' \wedge store \cong store_u \\
&\quad \textbf{if } expr = loc_i \wedge expr' = loc_j \Rightarrow i = j \wedge store \cong store_u \\
&\quad \textbf{if } reduceE \rightarrow reduceE' \wedge store \cong store_u \\
&\qquad \textbf{then } \exists unitTest'.\ unitTest \rightarrow\!\!\!\!\rightarrow_u unitTest' \wedge reduceE' \approx_u unitTest' \\
&\quad \textbf{if } unitTest \rightarrow_u unitTest' \wedge store \cong store_u \\
&\qquad \textbf{then } \exists reduceE'.\ reduceE \rightarrow\!\!\!\!\rightarrow reduceE' \wedge reduceE' \approx unitTest'
\end{aligned}
$$

**Theorem 8.2.** *reduceE* $\approx_u$ *unitTest is a weak bi-simulation.*

*Proof sketch by co-induction on* $\rightarrow$ *and* $\rightarrow_u$

This proof is similar to test-mode equivalence. However, for $\rightarrow_u$ to begin reduction with an isomorphic $store_u$, all object allocation for the test call must occur within $\rightarrow_u$ and not within the test specification. This can be achieved by creating an auxiliary object that builds the necessary data and supplies the properly annotated locations to the test specification expression. This does not reflect standard test development practice, but could be generated through an encoding of non-conforming programs. With a more flexible store isomorphism, we believe this restriction would not be necessary.

## 8.2 Trace complexity

Trace size can impact performance and data use, so restricting the growth of a trace increases the likelihood that a test execution will not negatively impact the ability of the program to run in a reasonable time and available space. As discussed previously, traces are reduced to limit the size of the data structure required and provide a flexible representation of the traversed program while retaining the correctness of the program flow. We use cyclomatic complexity [15] of the program as a bound on the size of our trace collection.

**Definition 8.8.** Cyclomatic complexity[15]

Let $G$ be the control-flow graph of the program, $N$ be the nodes or basic blocks of $G$, $E$ be the number of edges or branching points of $G$, and $P$ be the connected components of $G$.

Then $M$, the cyclomatic complexity of the program, is $M = E - N + 2P$.

Used in coverage analysis [20], cyclomatic complexity represents the number of distinct program paths in a program that a test suite achieving full coverage must follow. Note that our *trace* representation is also a graph of the basic blocks and branches within a program, where each *src* depicts an edge of the graph.

**Theorem 8.3.** *Trace upper bound*

*If* $g = controlFlowGraph(mid) \wedge \langle E[loc.mid()], store, \epsilon\rangle \rightarrow\!\!\!\!\rightarrow_u \langle E[value], store', t\rangle$ *then* $cyclomatic(t) \leq cyclomatic(g)$

*Proof sketch by induction on structure of method.*

## 8.3 Required value identification

Since we use trace annotations to recover traditionally unreachable objects, from the perspective of the method caller, we require confirmation that the annotation leads to a single value accessed at a uniquely distinguishable position. To gain this confidence, we examine the creation of the trace and its subsequent structure to establish the uniqueness of the annotation.

**Theorem 8.4.** *Unique introduction of* **req**

$\langle E[expr.mid()], store, \epsilon\rangle \rightarrow\!\!\!\!\rightarrow_u \langle E[value], store', trace\rangle \ \wedge (\textbf{req}\ id) \in trace \Rightarrow trace = trace' : src(\textbf{req}\ id) : trace'' \wedge (\textbf{req}\ id) \notin trace' \cup trace''$

*Proof sketch by induction on* $\rightarrow_u$.
$\mathbf{flatten}(trace') = t \Rightarrow \mathbf{stepsTo}(t, (\mathbf{req}\ id)) == \mathbf{stepsTo}(trace', (\mathbf{req}\ id))$
*Proof sketch by induction over trace structure.*

The first property demonstrates that if a (**req** *id*) annotation exists, then one, and only one, annotation for *id* is collected in a trace.

The flatten operation reduces iterations in a trace that can be construed as "counting" the times around the loop. The **req** annotation must exist along a path that can be generally repeated and identified, and cannot specify the $n$th identical iteration of a loop to extract the value (as the value of $n$ may vary on different executions). If the number of steps taken to the annotation on both the flattened and original trace are the same, then we must be traversing a unique path through the program and can identify the value.

# 9 Implementation

The semantics of Section 5 presents a view of test mode with little consideration of performance and other practical issues. In making a realistic implementation, performance and code-rewriting concerns require conservative optimizations. An implementation may not be able to report the theoretical maximum test errors when test overhead degrades performance unacceptably. However, with a flexible implementation, the performance degradation versus test results can be adjusted for differing systems and testing requirements by limiting the completeness of traces and the depth of object abstractions.

## 9.1 Representing snapshots and redirecting method calls

The snapshot class definitions presented in Fig. 8, which provide test-counterparts to method definitions, can be represented as classes which extend the tested class and override the specified methods. This follows the strategy we used previously in implementing transactional support for imperative methods. A call to an overridden method that contains test information must perform the semantic equivalent of CALLTM reduction for test mode.

The method body for test-mode snapshots calls a test engine representation which stores the pre-conditions for each method in a data base, and returns a set of accesses for the relevant post-conditions and required values. Each parameter, the embedded object, and any static variables known to be accessed by the method must have its snapshot nesting increased by one. Then the actual called method is invoked with the protected parameters and embedded object. After a successful return, the post-conditions selected by the test engine are checked through an appropriate method call, and the current nesting level of the transactions is committed before the value (where appropriate) is returned.

A Java method may not terminate naturally, and may instead be prematurely terminated with exceptions. We do not represent exceptions in the model to simplify the presentation, however this control flow change can be supported by our techniques. When exceptions may occur, we wrap the method call in a try block; since runtime exceptions are always possible, we pessimistically always require a try block. The commit for the method must occur in a finally block, to ensure that modifications are preserved for subsequent use. In our full post-condition specification language, we allow programmers to

check for exceptions as outlined in our previous work [11]. Only those post-conditions from the selected set containing these specifications are checked within the catch block of the expected exception. Each catch block must re-throw the exception.

In our unit-test implementation, we injected all objects into a snapshot at the method call boundaries for the tested method, or on their first read through an existing snapshot. For test mode, we can either perform this lazy building of snapshots, where nesting may first require inserting the object into a snapshot, or we can redirect constructor calls using an aspect-like rewriting of the program.

Super method calls, which do not dispatch through an object, may not therefore encounter test-method redirection. This must be solved by injecting an appropriate redirection through an aspect rewriting of such calls. Alternately, we must can these test calls and allow errors arising on super call boundaries to be ignored during system integration.

## 9.2 Trace implementation and trapping requirements

Traces are represented as objects, instances of a Trace class, with local state and hash tables tracking the previously visited locations. In collecting traces for the unit test, we require a globally visible data structure to collect the current trace which can be extracted and stored with the pre-condition. These traces must be serializable so that they can persist across multiple executions of the program. For test mode, there is only one trace to be collected of the program, which also must exist in a globally accessible static variable.

For method call traces, the source location for the entry and exit points could be controlled by the test method wrappers. However, this can cause a super call to be skipped, as discussed above. To capture branching behavior and super calls, we must insert calls to the tracing utility into each method, if branch, and looping construct. These modifications can be made through either using a test-aware compiler or on-the-fly rewriting of the byte codes.

In our theoretical model, upon entering a loop we stored the different (previously unseen) paths taken on each repetition. This may be too high a storage complexity for some programs. To optimize this situation, we can switch to more standard coverage storage within a loop or recursive method while retaining path information externally. However, this optimization reduces the percentage of required variables that can be identified with tracing and reduces the number of conclusive successful tests; so the optimization may be undesirable.

As in the operational semantics, variables declared with a `requirement` attribute are embedded into a specialized snapshot representation. For each required variable that is not a parameter, field of a parameter or static variable, we generate a *requirement* snapshot class that extends the appropriate snapshot class. These classes override each of the methods and field accessors in the snapshot to pass a flag, with an identifier, to the global trace object which identifies the access, when a local flag indicates this has not occurred before. After passing the access to the trace object, the local flag is set to false. The trace stores the located object and the identifier with the current position.

Upon reaching this position again during test-mode execution, the trace captures the current value and stores it for access during the post-condition execution. Each call to the trace object must provide an instance of the current object for all instance methods

to accommodate retrieving accessible values.

# 10 Related work: tests vs contracts

With pre- and post-condition validation combined with program execution, our system is reminiscent of traditional contract systems[17, 2, 5]. Influenced by test-driven development, we treat tests as a tool for exploring and guiding program development as well as validating correctness. Test executions are a distinct phase of development and execution, and tests are not part of the program. As such, test-mode execution differs from contract and assertion systems in four main ways.

A series of unit-test specifications express a conjunction of atomic Floyd-Hoare formulae: $\{P_1\}c\{Q_1\} \wedge \cdots \{P_n\}c\{Q_n\}$. However, many contract specifications are limited to a single pre- and post-condition pair (a notable exception is Ciao Prolog [16]). The above conjunction can be weakened to the Floyd-Hoare formula $\{P_1 \vee P_n\}c\{Q_1 \vee Q_n\}$, but with reduced expressivity as the relationship between $P_i$ and $Q_i$ is now lost; this weakening risks missing bugs identified by the original unit tests.

Contracts, though used to aid testing [4, 1], expect conditions to be clearly satisfied and typically halt execution if assertions fail. Although this identifies the source of system integration errors, such as those discussed in our introductory example, halting the program limits the amount of information to guide further development. If the program produces a valid result for the end user despite failing test specifications, then the programmers can determine if the test suite is in error instead of assuming the program does not work.

Since test-mode execution is not part of standard system execution, it is acceptable for tests to negatively impact performance to gather more information. And with tests existing outside the program, the test specification language will not impact existing programs or program deployment. Contracts and assertions must typically consider tradeoffs between increased costs and improved safety. The performance and memory overheads of transactional snapshots and trace-based correctness assessments may not be acceptable outside of a dedicated test-execution environment.

Finally, contracts and assertions evaluate correctness conditions immediately. As we show in Section 7, test specifications do not require this immediacy and thus can be used to gather non-localized correctness information.

Contracts can also be attached to individual objects or functions [7, 8]. These correctness properties are linked only to the particular value, not to the method definition. This usage of contracts is dissimilar to test specifications, which seek more general guarantees.

## 10.1 Other related work

Traditional coverage-based testing analysis [10] provides a metric for test quality. However, these metrics only identify untested paths in the program and do not consider the quality of a test otherwise. Even with full test coverage, errors can arise in system integration due to insufficient specifications in the co-operating unit tests.

Tools that examine execution paths during development can guide programmers in creating sufficient specifications [19, 13]. However, while these techniques help identify the specifications necessary for covering the program and satisfying the correctness conditions,

they do not demonstrate where correctness conditions are insufficient in an integrated system. By combining our approach with a similar technique to consider execution paths in an assisted environment, it may be possible to provide greater support for exploratory development with continuous condition checking.

# 11 Conclusions

The prevalence of unit testing and importance of test-driven development in building reliable software is growing. Yet bugs can be obscured by incorrect or non-exhaustive test specifications that build false confidence in the sub-systems of a program. With test mode, we expand the utility of unit tests to uncover software bugs and explore the reliability of system integration.

Our design and formalization lay the ground work for building stronger tools for test development. TestJava gives programmers a formal language for specifying test specifications without modifying existing test development strategies, and test mode provides greater confidence from unit tests along with information to assist in writing more rigorous correctness conditions. Test mode and test-mode equivalence also demonstrate that test executions can safely capture and defer program behavior without loss of confidence in the results.

While our model lacks real-world programming constructs such as exceptions and arrays, these features do not require significant modifications to the evaluation and data-capturing constructs we describe here. Extending support for multi-threaded programs should benefit from our prevalent use of transactions, but will require more work to properly specify correctness. In our forthcoming implementation, we add support for more specifications than represented here, including allowing predicates in requirement specifications as well as universal quantification over types, with concrete values optionally specified.

# References


[1] M. Barnett, M. Fähndrich, P. de Halleux, F. Logozzo, and N. Tillman. Exploiting the synergy between automated-test-generation and programming-by-contract. In *ICSE*, 2009.

[2] D. Bartezko, C. Fischer, M. Moller, and H. Wehrheim. Jass — Java with assertions. In *Workshop on Runtime Verification*, 2001.

[3] K. Beck. Simple smalltalk testing with patterns. *The Smalltalk Report*, 1994.

[4] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proc. ECOOP*, 2002.

[5] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM Software Engineering Notes*, 31, 2006.

[6] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the Java type system sound? *Theory and Practice of Object Systems*, 5, 1999.

[7] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proc. OOPSLA*, 2001.

[8] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proc. ACM International Conference on Functional Programming*, Oct. 2002.

[9] M. Flatt, S. Krishnamurthi, and M. Felleisen. A Programmer's Reduction Semantics for Classes and Mixins. In *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer, 1999.

[10] C. Gaston and D. Seifert. *Evaluating Coverage Based Testing*, chapter 11. Springer, 2005.

[11] K. E. Gray and A. Mycroft. Logical testing: Hoare-style specification meets executable validation. In *FASE*, 2009.

[12] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *Proc. OOPSLA*, 1999.

[13] L. Jiang and Z. Su. Profile-guided program simplification for effective testing and analysis. In *Proc. ACM Conference on Foundations of Software Engineering*, 2008.

[14] G. T. Leavens, A. L. Baker, and C. Ruby. *JML: A Notation for Detailed Design*, chapter 12. Kluwer, 1999.

[15] T. J. McCabe. A complexity measure. *IEEE Trans on Software Engineering*, SE-2, 1976.

[16] E. Mera, P. Lopez-García, and M. Hermenegildo. Integrating software testing and run-time checking in an assertion verification framework. In *Intern. Conf. on Logic Programming*, 2009.

[17] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. on Software Engineering*, 21, 1995.

[18] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: effective tool support for the working semanticist. In *Proc. ACM International Conference on Functional Programming*, 2007.

[19] N. Tillmann and J. de Halleux. Pex – white box test generation for .NET. In *Tests and Proofs*, 2008.

[20] A. H. Watson and T. J. McCabe. Structured testing: A testing methodology using cyclomatic complexity metric. In *NIST Special Publication 500-235*, 1996.