NST: A unit testing system for Common Lisp

John Maraist
SIFT, LLC
Minneapolis, Minnesota
jmaraist@sift.info

Appears in the *Proceedings of the 2010 International Lisp Conference*, Reno, Nevada, October 19–21.

ABSTRACT

We introduce NST, an expressive and extensible unit testing framework for Common Lisp. Distinct features of Lisp's object system CLOS allow an elegant implementation for NST, and we describe this class model and implementation. There are over a dozen test frameworks for Lisp which are currently available, almost half of which are under active maintenance, and we compare these packages' features. Development of NST is ongoing, and we conclude with a discussion of future directions for the system.

1. INTRODUCTION

In the last ten years the benefits of test-driven development have become very clear, and its adoption is now widespread. Many of its techniques are now widely accepted as best-practice software engineering standards:

- "Test first" practices such as translating bug reports and planned features into not-yet-passing tests.
- Use of testing to enforce design contracts for API functions.
- Evolution of regression tests suites which can be frequently, often automatically, rechecked to guarantee the correctness of a code repository.

With this paper we aim for three goals: First, we introduce NST, a new unit testing framework for Common Lisp. NST is an expressive, extensible system which can support a variety of testing styles. NST was developed as an in-house tool at SIFT, and has been available under an open-source license via *http://cliki.net/NST* since June 2009. Second, we provide a detailed comparison of current Lisp unit test systems. There are trade-offs of expressiveness and overhead among the various systems which give test authors many valid choices. Finally, NST's under-the-hood design offers ideas (or counterexamples!) to developers who find that no existing test framework serves their needs, and who therefore undertake their own testing framework. We begin in Section 2 with an overview of NST's features and syntax. NST translates the artifacts of testing to CLOS classes and objects, and we discuss this implementation in Section 3. We survey several unit test systems available for Lisp in Section 4. We conclude in Section 5 with a discussion of possible future directions for the project.

ILC '10 Reno, Nevada USA

2. SYSTEM OVERVIEW

NST shares many of the typical notions of unit test frameworks [15]. A *test* has three primary components: a name, a test *criterion*, and zero or more *forms under test*. *Execution* of a test assesses its forms in the manner specified by the criterion (which may or may not involve actually evaluating the forms). Each test is associated with a named *group*, allowing multiple tests to be executed by a single invocation of the group's name instead of each test's name; groups are also implicitly aggregated themselves under the Lisp package in which their symbolic names reside. Because many tests will tend to refer to a recurring collection of sample data, it is convenient to define named sets of *fixtures*, or test data, to provide consistent test environments. NST provides macros for defining tests, groups, fixtures and criteria.

2.1 Fixture sets

Essentially, fixtures are just a list of bindings of names to forms, as could be used in a let-binding, but not immediately associated with any particular scope.

```
(def-fixtures simple-fixture
    (:documentation "Define two bindings")
  (magic-number 120)
  (magic-symbol 'asdfg))
(def-fixtures more-fixtures
    (:special magic-number)
  (even-ints '(2 4 10 magic-number))
  (dodgy-even-ints '(1 2 4.0)))
(def-fixtures another-fixture ()
  (ordinary 3))
```

The second argument to the def-fixtures macro, on lines 2, 6 and 9 of the example, specifies options for fixtures via the usual Lisp convention for optional parameters. The examples show the documentation argument, and the special argument for declaring names expected to be bound for any later use of the fixture. Other options for def-fixture:

- Control whether the values bound when applying fixtures are evaluated once and cached for future applications, or re-bound at each application.
- Provide additional mechanisms for declaration over the scope of the fixtures' names.
- Specify code to initialize the fixtures, and clean up after it.

Bundling fixtures as a named set in this manner provides a number of benefits. First, we can expand them freshly as needed for different tests or groups, helping to isolate individual tests from each others' side-effects. We also avoid cluttering the package with global variables of test data, or alternately, needing a separate package to contain the clutter. At the same time, NST does provide the ability to inject these bindings as globals within the package for debugging.

2.2 Tests, groups and criteria

Groups and tests may be defined together, with tests implicitly included in the group within whose form the test definitions appear, or separately, with tests explicitly naming the group to which they are assigned:

```
(def-test-group simple-group (simple-fixtures)
  (def-test has-num ()
      (:eql magic-number)
    (factorial 5)))
(def-test has-sym (:group simple-group)
    (:eq magic-symbol)
  'asdfh)
(def-test-group other-group (simple-fixtures
                             more-fixtures)
  (def-test even-ints
      (:each (:all (:predicate evenp)
                   (:predicate integerp)))
    even-ints)
  (def-test even-int-trial
      (:each (:all (:predicate evenp)
                   (:predicate integerp)))
    dodgy-even-ints))
```

The group and test-defining macros accept additional options, including:

- Specify code to initialize the group or test, and clean up after it.
- Attach fixtures to individual tests.
- Add documentation strings.

2.3 Criteria

In the first two tests, (:eql magic-number) and (:eq magic-symbol) are *criteria* which specify what should be expected of the forms (factorial 5) and 'asdfh respectively. The second pair of tests shows how criteria can be combined and applied to more complex lists, structures and class objects. The :each criterion takes another *subcriterion* as its own argument, and expects its form under test to evaluate to a list. The overall criterion passes only if its subcriterion passes for each element of the evaluated list. The :all criterion expects that all of its subcriteria should pass on its forms under test. The :predicate criterion expects its argument, when applied as a function to the values returned by its forms under test, to return a non-nil value. The second test group also illustrates the use of multiple fixture sets: simple-fixtures and more-fixtures are applied in order, so that simple-fixtures provides a binding to the name magic-number, as more-fixtures requires.

The :eql, :eq, :each, :all and :predicate are some of the standard criteria defined within NST. Test authors can define and name their own criteria to conveniently perform arbitrary evaluations of forms. A simple example is suggested by the two tests of our other-group above: we can define a new criterion to abbreviate the repeated compound form,

```
(def-criterion-alias (:even-integer-list)
  `(:each (:all (:predicate evenp)
                (:predicate integerp))))
```

We can then simplify the group's test definitions,

```
(def-test-group other-group (simple-fixtures
                             more-fixtures)
  (def-test even-ints :even-integer-list
    even-ints)
  (def-test even-int-trial :even-integer-list
    dodgy-even-ints))
```

In addition to this "aliasing" mechanism for defining new criteria in terms of existing ones, NST also provides a more general mechanism akin to method definitions specifying the Lisp code mapping the criterion's arguments and the forms under test to NST's structures representing the details of test success or failure. We discuss criteria definitions and their implementation in Section 3.2 below.

At first glance, a separate sublanguage for criteria may seem excessive. For example, the Lisp predicate

```
(every #'(lambda (x)
           (and (evenp x) (integerp x)))
       FORM)
```

returns a true (non-nil) value exactly when the test

```
(def-test even-ints-form
    (:each (:all (:predicate evenp)
                 (:predicate integerp)))
  FORM)
```

passes. Especially for such a simple example, the overhead of a separate criteria language seems hardly justifiable. In fact, the criteria bring two primary advantages over Lisp forms. First, criteria can report more detailed information than just "pass" or "fail." Of course in the simple example of lists of numbers it will be obvious which members are not even integers! But in a larger application where the tested values are more complicated objects and structures, the reason for a test's failure may be more subtle. More informative reports can significantly assist the programmer, especially when validating changes to less familiar older or others' code. Moreover, NST's criteria can report multiple reasons for failure. Such more complicated analyses can reduce the boilerplate involved in writing tests; one test against a conjunctive criterion can provide as informative a result as a series of separate tests on the same form. As a project grows larger and more complex, and as a team of programmers and testers becomes less intimately familiar with all of the components of a system, criteria can both reduce tests' overall verbosity, while at the same time raising the usefulness of their results.

3. IMPLEMENTATION MODEL

In this section we give an overview of how NST works. We begin with a high-level view of NST's implementation model in Section 3.1. In Section 3.2 we discuss the implementation of NST's criteria. Finally in Section 3.3 we present further details of how NST supports key test framework features such as checking forms returning multiple values, setup and cleanup hooks, and trapping and reporting errors. We should point out that the implementation details are certainly not required knowledge for simply using NST. Throughout this section we will use the examples of the previous section as our running examples.

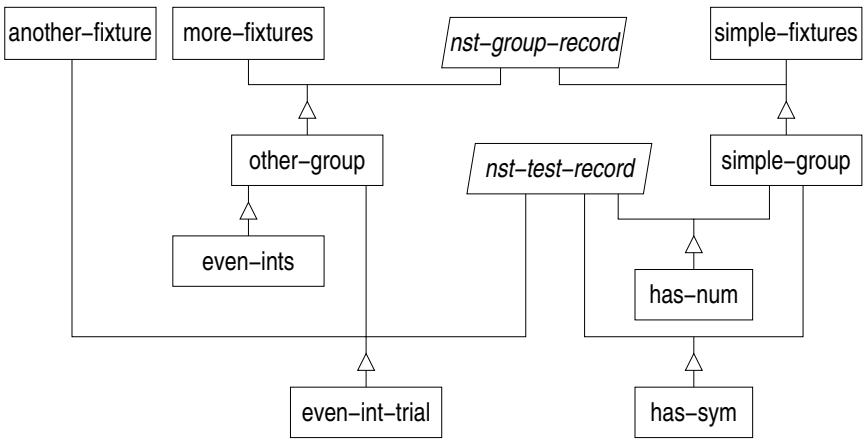


Figure 1: UML class diagram showing the class hierarchy of the translations of our running examples. The arrows link superclasses above to subclasses below.

```
(def-fixtures simple-fixture
    (:documentation
        "Define two bindings")
  (magic-number 120)
  (magic-symbol 'asdfg))
```

→

```
(defclass simple-fixture () ()
  (:metaclass singleton-class)
  (:documentation "Define two bindings"))
(defmethod do-group-fixture-assignment :around
                                ((g simple-fixture) ts)
  (declare (ignorable g ts))
  (let* ((magic-number 120)
         (magic-symbol 'asdfg))
    (declare (special magic-number magic-symbol))
    (call-next-method)))
(defmethod do-test-fixture-assignment :around
                                ((g simple-fixture) ts)
  (declare (ignorable g ts))
  (let* ((magic-number 120)
         (magic-symbol 'asdfg))
    (declare (special magic-number magic-symbol))
    (call-next-method)))
```

Figure 2: Translating fixtures into CLOS classes and methods.

```
(def-test-group simple-group (simple-fixtures))
```

→

```
(defclass simple-group (simple-fixtures)
    ()
  (:metaclass singleton-class))
```

Figure 3: Translating groups into CLOS classes and methods.

```
(def-test (has-num
           :group simple-group)
    (:eql magic-number)
  (factorial 5))
```

→

```
(defclass nst-intrnl::mypkg-simple-group--mypkg-has-num ()
    (nst-test-record)
  (:metaclass singleton-class))
(let ((tproto
       (make-instance
        'nst-intrnl::mypkg-simple-group--mypkg-has-num)))
  (flet ((set-slot (slot value)
           (setf (slot-value tproto slot) value)))
    (set-slot '%criterion '(:eql magic-number))
    (set-slot '%test-forms '((factorial 5)))
    (set-slot '%unique-name
          'nst-intrnl::mypkg-simple-group--mypkg-has-num)))
```

Figure 4: Translating tests into CLOS classes and methods.

3.1 Class model and main execution control

Fixtures, groups and tests all map to CLOS class definitions. Figure 1 shows the relationship among the classes to which our running examples would be translated. Fixtures translate to a class and two methods. One method is used to provide the fixture's definitions to all tests of a group when the group is invoked for testing; the other, to provide the definitions when an individual test is run. Figure 2 shows the translation of the simple-fixture set from our earlier example.[1] When a group or a test incorporates fixtures, the classes to which the fixtures translate become superclasses of the class to which the group or test is translated. This allows the group or tests to call methods which produce dynamic bindings corresponding to the fixture's definitions without requiring additional code to be generated for the bindings. Note that we declare the fixture's class, as well as the classes arising from groups and tests, to be a *singleton* class: only one instance of it will be created, simplifying the bookkeeping of tests and results. Of course this restriction is straightforward to implement with Lisp's metaobject protocol; a method for passing classes based on singleton-class to the generic function make-instance can detect and return an existing instance, or create the first instance with call-next-method.

Figure 3 shows the translation of a group definition. The crucial detail of this translation is that the new class corresponding to the group has as superclasses all of the groups corresponding to the fixture sets applied to the group. It is exactly this inheritance which allows the scope of the bindings of do-group-fixture-assignment to transfer to the group.

Tests have a common superclass nst-test-record,

```
(defclass nst-test-record ()
    ((%criterion   :reader test-criterion)
     (%test-forms  :reader test-forms)
     (%unique-name :reader test-unique-name)))
```

The three internal slots %criterion, %test-forms and %unique-name are initialized by the expansion of the def-test macro, as we illustrate in Figure 4. Note that they are stored as forms: by waiting to evaluate their syntax we require less recompilation — and in most cases, a more automatically-managed recompilation — to correctly test forms involving macros.

Unlike the translation of fixtures and groups, the translation of tests maps to a class with a different name than the original test. This translation allows test authors to give the same name to two tests in different groups but in the same Lisp package. Without the alteration of the name, the classes underlying two such tests would overwrite each other; requiring the user to ensure unique names for all tests in different groups of the same package seems onerous. The new name, which is in a package internal to NST, incorporates the package and symbol name of both the group and the test. Note that we do not gensym the new test name: we do want subsequent redefinitions of a test to redefine the same class.

With this class structure established, the main control for test execution is straightforward. Figure 5 contains an excerpt of these routines, illustrating the interaction of the

[1]For readability in examples of translations, we write formal parameters such as g and ts as ordinary names. In fact, per standard macro hygiene practices, these names are introduced via gensym to prevent the possibility of name capture.

```
(defun run-group-tests (group-obj test-objs)
  (do-group-fixture-assignment group-obj
                               test-objs))

(defgeneric do-group-fixture-assignment
                         (group-obj test-objs)
  (:method (group-obj test-objs)
    (loop for test-inst in test-objs do
      (do-test-fixture-assignment test-inst))))

(defgeneric do-test-fixture-assignment
                                      (test-obj)
  (:method (test-obj)
     (core-run-test test-obj)))

(defun core-run-test (test)
  (let* ((criterion (test-criterion test))
         (forms (test-forms test))
         (result
          (check-criterion-on-form
             criterion
             `(multiple-value-bind
                          #'list ,(car forms)))))
    (setf (gethash (test-unique-name test)
                   +results-record+)
          result)))
```

Figure 5: NST test execution.

classes underlying fixtures, groups and tests.[2] The entry point to this kernel is the run-group-tests function. Its first argument is a group object, and its second argument is a list of test objects: passing a collection as the second argument allows us to have a single entry point to the code whether running one, some or all of the tests in a group. The two generic functions whose primary methods are part of the kernel are the same two for which the translation of fixtures provides :around methods: since groups and tests are translated as subclasses of the fixtures which they use, the scope of these dynamic bindings will cover any reference in the criteria or in the forms under test. The core-run-test function applies the actual test criterion to the forms under test. It calls the check-criterion-on-form function — which as we will see below is part of the NST API for criteria definitions, used not only here but also when defining compound criteria such as :each — and stores the result in NST's internal hash table +results-record+ for test results.

3.2 Test criteria

A criterion specifies a mapping from two sources, the data over which the criterion itself may be parametrized and a form which evaluates to the values under test, to a report on the success or failure of the test. In general, a criterion receives its inputs as unevaluated forms. There are a number of motivations behind delaying these forms' evaluation: First, some of these data, such as subcriteria, should never be evaluated as-is. Moreover, the delay allows faithful support of the traditional Lisp-style of development, where the programmer redefines individual functions, structures and variable bindings, re-checking tests frequently. Specifically, evaluating each tests' forms at every test invocation allows a more seamless assessment of forms involving macros or otherwise in-lined calls, changes to which might not be in-

[2]The excerpt omits the details of various additional features, some of which we discuss in Section 3.3 below.

corporated in subsequent tests unless (inconveniently) the test itself is recompiled. Just as importantly, providing the original forms allows a criterion to more closely control the environment in which the forms' evaluation takes place. An obvious example is for observation of Lisp's errors and warnings, for which a criterion could establish a dynamic scope of handlers for particular conditions arising from form evaluation. We discuss NST's and its criteria's error handling in more detail in Section 3.3.3 below. The utility of passing forms to criteria is not limited to condition handlers; criteria can usefully alter, or provide a temporary dynamic binding for, global parameters, simulating various test situations.

3.2.1 Criteria aliasing

The simpler criteria-defining mechanism, which we introduced in Section 2.3, does not refer directly to the forms under test, but instead merely describes how a new criterion can be rewritten as existing criteria. The earlier example, :even-integer-list also did not receive any arguments with the criterion; a simple example of a criterion which does receive an argument is :symbol:

```
(def-criterion-alias (:symbol name)
  `(:eq ',name))
```

The :symbol criterion simply rewrites to the :eq criterion, adding the symbol's quotation mark.

3.2.2 General criteria definitions

Since a criterion receives both a form which evaluates to the values under test and the data over which the criterion itself may be parametrized, our general def-criterion macro defines a criterion's method in terms of not one, but two lambda lists:

```
(def-criterion (CRITERION-NAME CR-ARGS-LLIST
                               TEST-VALS-LLIST)
  FORM
  ...
  FORM)
```

Criterion arguments will sometimes include subcriteria or other forms which should not be evaluated; criteria will sometimes need to wrap the evaluation of the forms under test in an error handler or other dynamic scope. On the other hand, when a criterion does not require unevaluated arguments from one or the other lambda list, it is convenient to have some method for eliding the boilerplate of calling eval and destructuring its results. For this reason, def-criterion allows the test author to choose between call-by-name and call-by-value parameter-passing semantics for each of its lambda list. For example, we might define the :eq criterion as:

```
(def-criterion (:eq (:values target)
                    (:values actual))
  (cond
   ((eq target actual)
    (make-success-report))
   (t
    (make-failure-report
      :format "Not eq to ~s: ~s"
      :args (list target actual)))))
```

The symbol at the head of CR-ARGS-LLIST and TEST-VALS-LLIST specifies the parameter-passing semantics, and the remainder of each list is a lambda list. An alternative definition for :eq which provides better failure information is:

```
(def-criterion (:eq (:forms target-form)
                    (:form actuals-form))
  (destructuring-bind (actual)
      (eval actuals-form)
    (let ((target (eval target-form)))
      (cond
        ((eq target actual)
         (make-success-report))
        (t
         (make-failure-report
           :format "~s evaluates to ~s, not ~
                    eq to ~s from ~s"
           :args (list actuals-form actual
                       target target-form)))))))
```

There is a subtle distinction between the unevaluated modes of the two lambda lists. While unevaluated criterion arguments are matched against a general lambda list just as for macro expansion, when the forms under test are passed unevaluated *they are passed as a single form, which can be bound only to a single formal parameter.* Like any Lisp expression a form under test may return multiple values; NST ensures that multiple values are converted to a list (which we discuss in Section 3.3.1 below), so that the result of evaluating the form can be easily deconstructed.

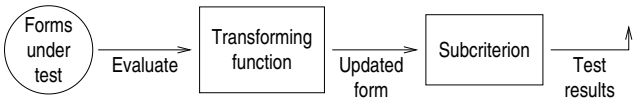
The :eq criterion definitions also show the use of the make-success-report and make-failure-report functions. The criterion is expected to return an NST-defined structure that captures the contexts of errors, failures and warnings; these functions are the NST test author's API for creating these structures. Instead of simple pass/fail/error reporting, these structures encapsulate lists detailing the (possibly many) reasons for a test's failure. Along with the lists of failures and errors, NST also accumulates runtime warnings, plus informational messages which can be generated by test criteria. This approach to reporting (hopefully empty) lists of problems instead of a single success is by no means novel; it is a variation of the technique first described by Wadler [29]. Here, applying this pattern allows NST to report the more detailed, hopefully more useful, test results for which we argued in Section 2.3.

3.2.3 Recursive application of subcriteria

The NST criteria which we have discussed so far in this section all perform simple tests of scalar values. However, more complex criteria are possible: these criteria contain *subcriteria* which apply to transformations of the originally provided values, or to components of class or structure objects. NST provides two functions which facilitate a recursive criterion invocation. We have already seen one of these functions, check-criterion-on-form, called by core-run-test in Figure 5 above. Both this function and check-criterion-on-value provide thin wrappers to the generic function apply-criterion: NST's criteria map directly to eql-methods of this function. The methods dispatch on the symbol naming the method, and receive two additional parameters: the list of forms given as arguments to the criterion, and a form which, when evaluated, returns the values under test. The check-criterion-on-form and check-criterion-on-value functions simply unpack the criterion names and arguments, and in the latter case prepare the already-evaluated value to be passed to this underlying function which does not necessarily expect it to have been evaluated:

```
(defun check-criterion-on-form (criterion form)
  (unless (listp criterion)
    (setf criterion (list criterion)))
  (apply-criterion (car criterion)
                   (cdr criterion) form))
(defun check-criterion-on-value (criterion val)
  (check-criterion-on-form criterion
                           `(list ',val)))
```

The :apply criterion provides a simple example of a recursive criterion application. With :apply we can define a new criterion by applying some transformation to the original forms under test, and assessing them under some other subcriterion:



We can define :apply as:

```
(def-criterion (:apply (:forms transform
                               criterion)
                       (:form form))
  (check-criterion-on-form criterion
    `(multiple-value-call #'list
        (apply #',transform ,form))))
```

For example, we might use the :apply criterion in this test:

```
(def-test a1 (:group g1)
    (:apply max (:predicate zerop))
  -1 -10 (- 5 5))
```

The :predicate criterion is another built-in criterion, which uses a Lisp function. So when core-run-test processes test a1, it applies the criterion

```
(:apply max (:predicate zerop))
```

to the unevaluated forms

```
'(-1 -10 (- 5 5))
```

using check-criterion-on-form. Following the definition of :apply, Lisp first evaluates

```
(apply #'max (list -1 -10 (- 5 5)))
```

to the value 0. This value is then checked by the (:predicate zerop) criterion, which passes.

3.2.4 Implementing criteria

As we mentioned in the previous section, all criteria definitions are translated to methods of the generic function apply-criterion. Figure 6 illustrates the translation generated by each of the three criteria-defining mechanisms discussed above. Translations of the different parameter-passing semantics of def-criterion differ only in that the call-by-value modes include an evaluation of the forms under test, matching the results of that evaluation instead of the unevaluated forms to the given lambda lists. Criteria defined using def-criterion-alias are just translated to a def-criterion which uses call-by-name semantics for the forms under test, reflecting that these alias definitions do not directly mention, and thus should not compel the evaluation of, the forms under test.

3.3 Implementation of other key features

The class model and core routines we presented in Section 3.1 are a considerable simplification of the actual NST code. To more clearly illustrate the overall structure, we have elided details of many major features as well as all mundane details of bookkeeping. In this section we describe the implementation in NST of three key Lisp unit testing capabilities: cleanly handling Lisp's ability to return multiple values from a function; specifying routines to set up tests and clean up after them; and handling errors and warnings encountered during test execution.

3.3.1 Handling multiple values

Lisp functions are able to return multiple values. In most usage contexts all but the first value is ignored; the programmer must apply additional structure to access the additional results. In designing NST's handling of multiple-valued results, we reverse this philosophy: NST expects all of the values returned from a form to be tested; the test author must use additional structure to *ignore* the additional results. Lisp's convention emphasizes brevity for common use cases, but NST's emphasis is on the comprehensiveness of testing: *use* of some function should certainly be easy and concise, but good *testing* practice should look at all of the results, not just the primary one.

In the apply-criterion function, the third argument is a form which, when evaluated, returns a list containing the multiple results. In Figure 6, we see an example of how def-criterion matches against this list: the lambda list specifying the values under test becomes the lambda list of the deconstructing-bind applied to the result of evaluating the incoming form. For convenience, the test author can explicitly enumerate multiple values in a def-test; to support this variation we amend the translation of def-test illustrated in Figure 4 as follows:

```
(defun core-run-test (test)
  (let ((criterion (test-criterion test))
        (forms (test-forms test)))
    (cond
     ((eql 1 (length forms))
      (check-criterion-on-form
         criterion
         `(common-lisp:multiple-value-list
             ,(car forms))))

     (t (check-criterion-on-form
           criterion `(list ,@forms))))))
```

3.3.2 Setup and cleanup routines

In order to consistently recreate accurate test environments, NST supports several ways of defining *setup* and *cleanup* hooks:

- A hook can be defined for the start (or end) of a fixture set, running whenever the fixture is applied to a group or test.
- A hook can be defined for the start (or end) of a particular test.
- A hook can be defined for the start (or end) of a group of tests, running before the first (after the last) test of the group. Group hooks happily coexist with test hooks; a test's hooks bracket the execution of the test, and are themselves bracketed by the hooks of the group to which the test belongs.
- A fixture set accepts separate pairs of hooks for setup/-cleanup within the scope of their fixture sets, and for before or after the names are in scope. Groups and tests accept two similar pairs of hooks.

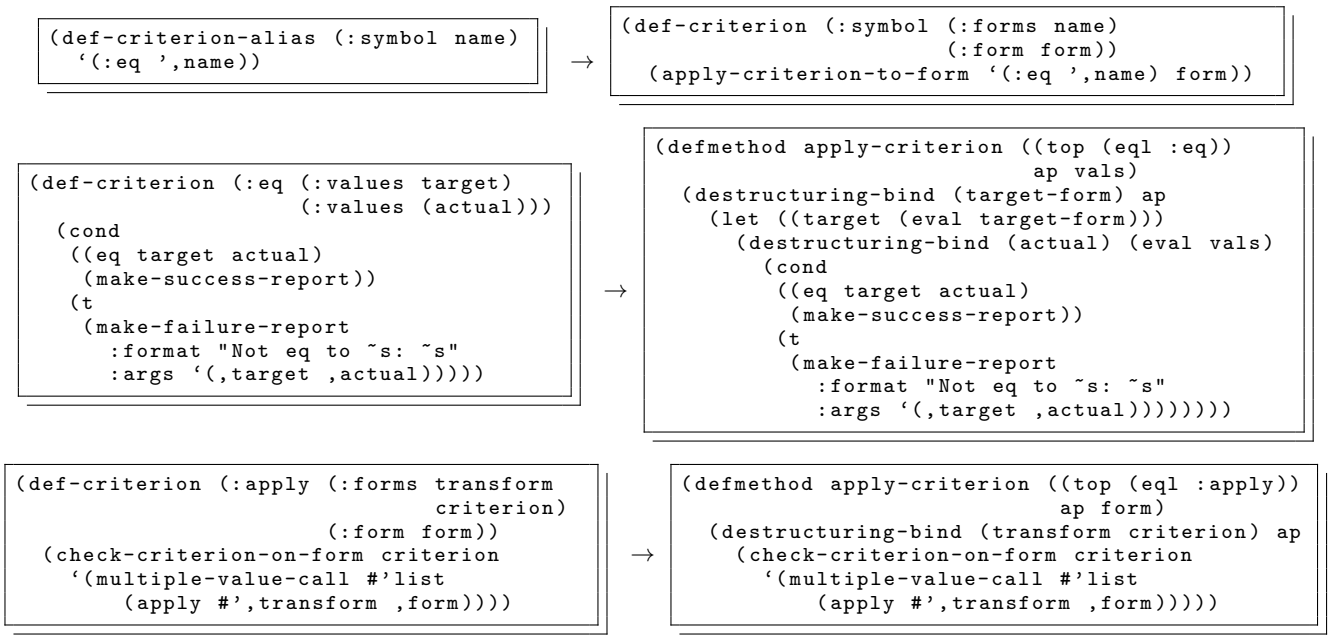


Figure 6: Translating criteria definitions.

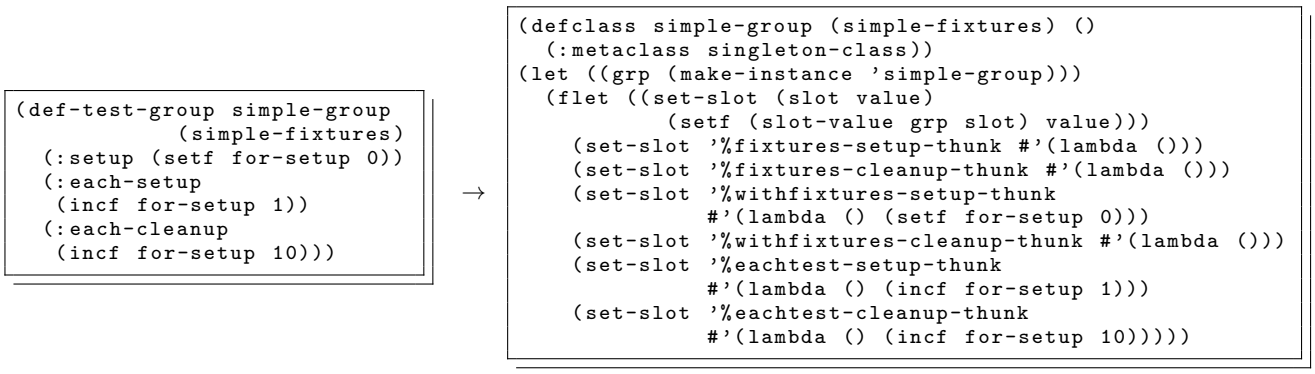


Figure 7: Translating a group with setup/cleanup hooks.

- Finally, a hook may be provided with a group's definition to run before (after) each individual test of the group.

These hooks are useful for configuring test environments, for example, when opening (and later closing) a database connection before binding fixtures extracted from the database. Where we have a single database of test values, we might use that fixture set's hooks to manage the database connection. Where we have multiple, similarly-configured databases, we might reuse one fixture set but manage the database connection with hooks on groups or tests.

NST traps errors thrown from hooks. Where the user has activated debugging on errors, NST offers restarts for ignoring the error and proceeding with testing, skipping the current test of group, or aborting testing (optionally skipping or running the various cleanup hooks when the error occurs earlier). We discuss NST's error handling in more detail in the next section.

Implementing these hooks is straightforward; Figure 8 shows extensions to the previous definitions for the classes nst-test-record and nst-group-record, and for the do-group-fixture-assignment function. The nullary functions corresponding to the hooks live in slots defined in nst-test-record and nst-group-record. These slots are set in the translation of def-test-group and def-test forms; Figure 7 shows the extended translation of a test group. These slots are read and the functions executed by the core routines of Figure 5. For example, the do-group-fixture-assignment function invokes the group setup/cleanup function for within group fixtures' scope, the setup/cleanup associated with the group for each of its tests, and the test setup/cleanup for outside the test fixtures' scope.

3.3.3 Handling errors

One essential capability for a unit test framework is that it can capture and report not only failures to satisfy test criteria, but also runtime program errors. The interactive nature

```
(defclass nst-group-record ()
  ((%fixtures-setup-thunk   :reader group-fixtures-setup-thunk)
   (%fixtures-cleanup-thunk :reader group-fixtures-cleanup-thunk)
   (%withfixtures-setup-thunk   :reader group-withfixtures-setup-thunk)
   (%withfixtures-cleanup-thunk :reader group-withfixtures-cleanup-thunk)
   (%eachtest-setup-thunk   :reader group-eachtest-setup-thunk)
   (%eachtest-cleanup-thunk :reader group-eachtest-cleanup-thunk)))
(defclass nst-test-record ()
  ((%test-forms :reader test-forms)
   (%criterion  :reader test-criterion)
   (%prefixture-setup    :reader prefixture-setup-thunk)
   (%postfixture-cleanup :reader postfixture-cleanup-thunk)))
(defgeneric do-group-fixture-assignment (g tsts)
  (:method (g tsts)
     (funcall (group-withfixtures-setup-thunk g))
     (loop for tst in tsts do
       (funcall (group-eachtest-setup-thunk g))
       (funcall (prefixture-setup-thunk tst))
       (do-test-fixture-assignment tst)
       (funcall (postfixture-cleanup-thunk tst))
       (funcall (group-eachtest-cleanup-thunk g)))
     (funcall (group-withfixtures-cleanup-thunk g))))
```

Figure 8: Implementing hooks.

of Lisp development places additional, unique burdens on a Lisp test framework. Lisp developers are best supported by providing both:

- The batch ability to run all tests without pausing, with a report on errors and failures at the end, as well as
- Support for interactive development, where errors (or failures) cause a break to the Lisp REPL, allowing the developer to examine the dynamic environment, and to choose to restart or skip tests and groups.

NST's treatment of errors at testing time is governed by the global flag *debug-on-error*.[3] If it is non-nil, then on intercepting an error NST should enter the debugger, allowing the user to examine the test state, and possibly restart tests. On the other hand, if *debug-on-error* is nil, then NST should note the error but continue with testing, reporting the error with its results afterward. Consequently much of NST's error processing branches on this flag.

There are three source of errors in NST test runs:[4]

1. Errors in evaluating the forms under test.
2. Errors in evaluating fixtures, setup, and cleanup.
3. Errors arising from test criteria coding.

The latter two of these sources reflect the reality that secondary testing code is as subject to bugs as primary application code! It is important for the framework to correctly characterize the source of an error in test execution.

To catch errors in user setup/cleanup routines, we wrap the various funcalls to the various funcalls invoking these thunks:

- When testing interactively, to offer restarts allowing either:
  - The routine to be re-run,
  - Tests to proceed despite the setup/cleanup error, or
  - The test or group to be skipped.
- For setup routines, to record the error as a test failure when not retrying.
- For cleanup routines, to note the error along with the test result when not retrying.

The NST source uses macros to tame the boilerplate code arising from the repetition of these patterns of handlers and restarts arising at every funcall, while still allowing small differences as each instance requires. To process errors arising from fixtures, we add similar mechanisms for restarts and reporting within the let*-bound forms of the :around methods of do-test-fixture-assignment and do-group-fixture-assignment illustrated in Figure 2, which allow the names of the erring fixture and its fixture set to be reported to the test runner.

Distinguishing errors of types 1 and 3 can be tricky. The availability of the unevaluated forms under test to criteria, combined with the freedom which test authors have to define arbitrary criteria performing arbitrary manipulations to the forms under test, means that either of these types of error could arise from the same code. The illustrations of criteria definition expansions in Figure 6 show how the sources of these errors may alternate: strictly speaking the only sure source of errors arising from the forms under test is from the (eval vals) call, but this expression occurs between error sources which could be considered configuration errors from either the criterion's coding, or its application. NST provides two macros returning-test-error and returning-criterion-config-error which generate a test result object appropriate to the respective error sources. NST inserts calls to these macros within the code generated from def-criterion. For example, in Figure 9 we extend Figure 6's simplified translation of the :eq criterion.

4. RELATED WORK

In this section we survey a number of other available Lisp unit test systems, and compare them to NST. Table 1 summarizes the features of these systems.

General architecture.

XUnit refers to a structure of tests arranged into groups

[3]NST treats failures by a similar mechanism; we omit the repetitive details here. Note that these global variables are implementation aspects; NST's user interface via the REPL provides a mechanism for controlling its behavior that does not require users to manipulate its global variables.

[4]Aside from bugs in NST itself!

```
(defmethod apply-criterion ((top (eql :eq)) ap val-form)
  (returning-criterion-config-error
      ((format nil "Criterion arguments ~a do not match lambda-list (target)" ap args-formals))
    (destructuring-bind (target) ap
      (let ((vals (returning-test-error (eval val-form))))
        (returning-criterion-config-error
            ((format nil "Values under test ~a do not match lambda-list (actual)" vals))
          (destructuring-bind (actual) vals
            (returning-criterion-config-error ("Error from criterion :eq body")
              (cond
               ((eq (eval target) actual)
                (make-success-report))
               (t (make-failure-report
                    :format "Not eq to value of ~s: ~s" :args `('target ,actual)))))))))))
```

Figure 9: Extending criteria expansion for error tracking.

or suites in the manner of Beck's proposal for Smalltalk [6] and its descendants such as JUnit. Tests and groups are typically all named, and have methods for configuring their environment. CLOS-unit [23, 22], HEUTE [21] and XLUnit [9] are xUnit style frameworks intended as a literal reimplementation of JUnit in Lisp [23].[5] They hew very closely to the object-oriented style of the original xUnit frameworks (which we indicate as *OO* in the table): their equivalent of a group is a defclass definition with a particular superclass. To define tests, one simply defines methods on this class with def-test-method, an enhanced defmethod. For example, to express the example tests of Section 2 we could write:[6]

```
(defclass simple-group (test-case)
  ((magic-number :accessor magic-number)
   (magic-symbol :accessor magic-symbol)))
(defmethod set-up ((g simple-group))
  (setf (magic-number g) 120
        (magic-symbol g) 'asdfg))
(def-test-method has-num ((g simple-group))
  (assert-equal (magic-number g)
                (factorial 5)))
(def-test-method has-sym ((g simple-group))
  (assert-true (eq (magic-number g) 'asdfh)))
```

Most of the available Lisp unit test systems are in the general xUnit style, but provide macros which more completely hide the underlying object model of the test framework. NST is of this style (which we mark *F* for "functional" in the table), as are FiveAM [4], LIFT [17, 18], Stefil [14, 19], lisp-unit [14, 25], CLUnit [1], and MSL.TEST [11]. The basic forms for defining groups and tests in these systems are generally similar; for example in LIFT we would write the simple-group as:

```
(deftestsuite simple-group ()
    ((magic-number 120)
     (magic-symbol 'asdfg))
  (:tests
    (has-num
      (ensure-same (factorial 5) magic-number))
    (has-sym
      (ensure-same 'asdfh magic-symbol))))
```

[5]An additional system XPTEST seems to have been superseded by XLUnit; its URL listed on CLiki *et al.* no longer exists, although periodic updates to its source are made to a Debian-distributed package (which does not include documentation).

[6]We use XLUnit for our examples of this style, although all three use very similar syntax.

We also survey two additional systems RT [26, 30] and Franz's tester [12] which are much simpler than the xUnit architecture.

Correctness criteria.

Every test system includes some way of describing the "correct" result of a test, which is the table we summarize in the columns under the *Criteria* heading. The various test frameworks fall into three general groups of what specifications the system allows, which we summarize in the *Form* column.

- t A test is just an expression to be evaluated; non-null values indicate a passing test. RT, CLOS-unit and tester specify correct behavior this way.
- S Tests specify atomic criteria on scalars, but the criteria do not include subcriteria for components of data structures such as lists, structs, and objects. FiveAM, Stefil, LIFT, XLUnit, lisp-unit, MSL.TEST, CLUnit, and HEUTE offer simple criteria.
- C Tests specify complex criteria, involving sub-criteria for testing structure components. NST provides complex criteria.

NST and LIFT also provide facilities within their frameworks to *extend* the set of test criteria.

We argued for the advantages of complex criteria in Section 2.3. Systems where test objects are explicitly exposed offer a possible alternative mechanism for achieving similar ends. For example, XLUnit *et al.* do not export a facility for extending their criteria, but this restriction is not necessarily a straightjacket. Their more explicit exposure of test class definitions allows us to parametrize particular instances over the individual objects which we would like to test according to a particular rubric encoded as a subclass of test-case.

```
(defclass even-integer-list (test-case)
  ((the-list :initarg :the-list
             :reader the-list)))
(def-test-method is-even-integer-list
                        ((g even-integer-list))
  (assert-true
    (every #'(lambda (x) (and (evenp x)
                              (integerp x)))
             (the-list g))))
```

Unfortunately, XLUnit's exported macro get-suite takes only the class name of a test-case, not any keyword arguments which might be passed to initialize-instance — but this omission is easily patched in a local installation, at least for

manual use in the REPL on an instance-by-instance basis. Similarly, LIFT's `deftestsuite` macro generates similarly-named initarg keywords for each slot, and its API functions run-test and run-tests do in fact accept a testsuite-initargs argument. These features suggest[7] allowing a LIFT test suite to be applied as NST applies criteria, harvesting multiple, separately reported tests abstracted over the form of interest.

```
(deftestsuite even-integer-list () (this-list)
  (:test
    (ensure-true
      (every #'(lambda (x)
                 (and (evenp x)
                      (integerp x)))
             this-list))))
```

However, in both of these cases it is not clear how to naturally incorporate a series of applications of one suite to different values into another suite. Furthermore, while these workarounds do mimic NST criteria's abstraction of multiple tests, the limitation that each individual test stops at the first failure remains: there is no analogue to NST's :each criterion which will ensure that *every* element of the list passes some subcriterion. In other words, these techniques apply well to multiple tests along the *static* structure of complex objects under test, but they account less well for *dynamic* variations of inputs.

Individual tests.

The columns under the *Tests* heading describe how individual tests are phrased. Note that terms including "test" are used in different ways by different systems; in this comparison and table we describe the functionality denoted by our use of each term in NST. So by "test" we mean the application of one test criterion to a form or forms, regardless of whether a particular platform uses "test" to describe such an artifact, or something else.

In the current versions of almost all of the platforms, a test on a form including a macro will not need to be explicitly recompiled or reloaded when the definition of the macro changes (possibly by setting a flag for that test). The *Macro* column reflects these implementations; tester does not store tests for later re-execution, so this concern is not relevant to that package.

The *Invoke* column concerns how tests are run. Almost all systems allow tests to be (re-) run by name. In lisp-unit and MSL.TEST, basic tests are not named; only groups of tests are named, and these groups can be explicitly invoked. Tester does not name tests, nor allow groups to be invoked by name, so they can be re-run only by re-entering them on the REPL, or re-loading the file which contains them.

Deleting tests is useful when developing tests interactively; an author may decide that a test is incorrect or inappropriate. Of course, test deletion is relevant only when tests are re-runnable. NST, LIFT, lisp-unit, CLUnit and RT allow tests to be deleted.

Test groups/suites.

The columns under the *Groups* heading enumerate features available for test groups. Most simply, a check-mark in the *Definable* column indicates whether some sort of test grouping is available. The original release of RT does not allow tests to be grouped, although some packages branching it do. Many systems allow test groups to be nested, so that invoking one group checks not only that group's own tests, but also a set of other groups identified with as the first group's "children." FiveAM and HEUTE allow such nesting of groups within each other; NST and lisp-unit offer a weaker form of nesting where the groups within a particular package can be invoked. In the *Nestable* column, a check-mark indicates that nested groups are generally allowed; a "P" indicates nesting of groups within a package, but not within other groups. A check-mark in the *Delete* column indicates that the framework provides a mechanism for deleting test groups. Finally, a check-mark in the *Test dep.* column indicates that a framework allow the execution to be ordered and dependent on whether other tests pass or fail; only FiveAM and MSL.TEST offer this feature.

Fixture sets, and setup and cleanup hooks.

The columns under the *Fixtures* heading describe platforms' facilities for naming sample data referred to in tests. A number of test packages use the term "fixture" to describe the facility for the general setup and cleanup routines of tests or groups of tests; we consider specifying name-form binding lists from these hook functions separately.

The *Definable* column describe whether fixture sets can be defined through the test framework. NST, FiveAM and LIFT allow sets of fixture bindings to be named and re-used. LIFT does not model these bindings as a separate entity; binding are associated with a test suite, and are inherited by any suite extending the suite defining the fixtures. Revisiting LIFT's version of simple-group, we can factor out the fixture set definitions,

```
(deftestsuite simple-fixtures ()
    ((magic-number 120)
     (magic-symbol 'asdfg)))
(deftestsuite simple-group (simple-fixtures) ()
  (:tests ...
```

The documentation of Stefil, CLOS-unit and XLUnit describe how their setup and cleanup hooks can be harnessed to provide fixtures (indicated by an "L" in the table). These mechanisms tend to introduce fixtures via accessors rather than simple bound names, as we saw in the XLUnit version of simple-group above.

A check-mark in the *Groups* column indicates that fixture sets may be applied to groups; in the *Tests* column, to individual tests.

The columns under the *Hooks* heading describe how the platforms allow setup and cleanup routines. A check-mark in the *Fixtures* column indicates that setup and cleanup hooks may be applied to fixture set uses; in the *Groups* column, to test groups; in the *Tests* column, to individual tests.

Results.

The columns under the *Results* heading describe the platforms' facilities for storing, retrieving and inspecting test failure and error reports. A check-mark in the *Stored* column indicates that test results are stored by the platform, as opposed to simply printed once and then discarded.

There are no check-marks in the *History* column: no platform records the history of test runs in some persistent storage, and allows the results over time to be examined. FReT [20] is an in-progress project with a slight variation of the LIFT API. Because it seems to still be working to-

[7] The testsuite-initargs argument is undocumented.

	Criteria			Tests			Groups				Fixtures			Hooks			Results		
XUnit	Form	Extend	Macro	Invoke	Delete	Definable	Nestable	Delete	Test dep.	Definable	Groups	Tests	Fixtures	Groups	Tests	Stored	History	Open
NST[1]	F	C	✓	✓	✓	✓	✓	P	✓	×	✓	✓	✓	✓	✓	✓	✓	×	×
FiveAM[2]	F	S	×	✓	✓	×	✓	✓	×	✓	✓	✓	×	×	×	×	✓	×	×
Stefil[3]	F	S	×	✓	✓	×	✓	×	×	×	L	×	✓	—	×	✓	✓	×	×
LIFT[4]	F	S	✓	✓	✓	✓	✓	×	×	×	✓	✓[5]	✓	—	✓	✓	✓	×	×
lisp-unit[6]	F	S	×	✓	×[7]	✓	✓	P	✓	×	×	—	—	×	×	×	×	×	✓
MSL.TEST[8]	F	S	×	✓	×	×	✓	×	×	✓	×	—	—	×	×	×	×	×	×
CLUnit[9]	F	S	×	✓	✓	✓	✓	×	×	×	×	—	—	×	×	×	×	×	×
RT[10]	S	t	—	✓	✓	✓	×[11]	—	—	—	×	—	—	×	×	×	×	×	×
CLOS-unit[12]	OO	t	—	✓	✓	×	✓	×	×	×	L	✓	×	—	✓	×	✓	×	✓
HEUTE[13]	OO	S	×	✓	✓	×	✓	✓	×	×	×	—	—	—	✓	✓	✓	×	✓
XLunit[14]	OO	t	—	✓	✓	×	✓	×	×	×	L	✓	×	—	✓	×	✓	×	✓
(p) tester[15]	S	t	—	—[16]	×	—	✓	×	×	×	×	×	×	×	×	×	×	×	×

	Warnings	Qkc	GUI	ASDF	SLIME	**Last**
NST	✓	✓	×	✓	×	9/2010
FiveAM	×	✓	×	S	×	3/2008[17]
Stefil	×	×	×	S	✓	7/2009
LIFT	✓[18]	×[19]	×	S	×	4/2010
lisp-unit	×	×	×	S	×	Unav.
MSL.TEST	×	×	×	×	×	Unav.
CLUnit	×	×	×	×	×	11/2002
RT	×	×[20]	×	S	×	12/1990
CLOS-unit	×	×	×	×	×	8/2002
HEUTE	×	×	✓	×	×	1/2006
XLunit	×	×	×	S	×	9/2007[21]
(p) tester	×	×	×	×	×	6/2009
Clickcheck	×	✓	×	S	×	9/2005

General key:
✓ Available feature.
× Unavailable feature.
— Not applicable under the particular system architecture.
In particular columns:

- Under XUnit:
  - OO XUnit style in CLOS.
  - F XUnit in functional style, hiding underlying CLOS implementation.
  - S Simpler model than xUnit.
- Under From:
  - t t/nil testing.
  - S Named criteria on scalars.
  - C Named criteria on structured objects.
- Under Nestable:
  - P Via package system only.
- Under Fixtures-Definable:
  - L Following documented examples of CLOS methods.
- Under ASDF:
  - S For loading the test system itself only.

[1] Version 2.1.1.
[2] [4, accessed August 2010].
[3] [19, accessed August 2010].
[4] [16, accessed August 2010]. Some of LIFT's more recent features are undocumented [17, March 2008 and April 2010 news blurbs]. So while our table entries are at least up-to-date with LIFT's user guide, we may have overlooked some features present in the source code but not fully documented.
[5] The bindings for a group are applied separately to each test in the group.
[6] [25, accessed August 2010].
[7] Lisp-unit does not name its simple applications of criteria to forms under test. They use the word "test" as we use "group," to refer to several of these basic criteria applications.
[8] [11, accessed August 2010].
[9] [2, accessed August 2010].
[10] [26, accessed August 2010]. A number of extensions of RT exist; however these do not seem to be maintained.
[11] But groups are commonly supported in extensions of RT.
[12] [22, accessed August 2010].
[13] [21, accessed August 2010].
[14] Based on examples in code distribution [9, accessed August 2010]; no documentation distributed.
[15] [12, accessed August 2010]
[16] Tester does not store tests for later re-execution, so the four latter points of comparison of tests are moot.
[17] Date of last snapshot tarball.
[18] A global variable controls whether LIFT should treat runtime warnings as errors, and there is a test criterion which expects a warning to be raised.
[19] LIFT supports random generation of certain scalars, but not of more complex objects, and does not seem to support sampled verification of invariants.
[20] Dietz's extensions support randomized repetition of tests [10].
[21] Date of files in distribution via CLiki.

Table 1: Comparison of Lisp unit test packages.

wards an initial "1.0" release, we have omitted it from the summary table; however FReT is notable as the only one of these systems which proposes (though does not yet implement) a retrospective view of test results over time. More recently, developers' tools such as Hudson have provided a similar functionality in a non-language specific way.

When developing GUIs or other extensions which benefit from asynchronous access to events in the testing process, it is useful to find an open architecture for processing test results, for example by allowing Listener pattern [13] hook functions to be registered with the platform. Lisp-unit, CLOS-unit, HEUTE and CLunit make particular documentation of APIs for this purpose, indicated in the *Open* column.

Other features.

The *Warnings* column describes whether the system provides support for capturing runtime warning emitted by tests; note that we do exclude the trivial "support" of allowing a form which provides its own handler-bind/muffle-warning structure to catch and detect any warnings.

Quickcheck [7] is a lightweight framework for random generation of data for validating program properties. This style of testing is somewhat more complicated than specific tests on single, bespoke forms. There are two distinct aspects: describing how sample data is to be generated, especially for more complex objects, and specifying invariant properties on these data. Discussions of Quickcheck and its implementation are widely available,[8] and we do not discuss them further here. We summarize platforms' implementation of Quickcheck in the *Qkc* column. NST and FiveAM include an implementation of Quickcheck-style testing. Clickcheck is a standalone Quickcheck implementation which could be used in any test framework. LIFT provides some support for randomized values of particular types; Dietz's extensions to RT [10] support randomized repetition of tests.

Of the systems we considered, only HEUTE includes a *GUI* for displaying the progress and results of unit testing.

ASDF (Another System Definition Facility) is a commonly used protocol for packaging Lisp systems and for specifying how they should be loaded, run and tested [5]. Unit test systems can integrate with ASDF on two levels. More simply, a test system may be packaged so that it can be loaded via ASDF, perhaps as a dependency for testing another system, indicated in the table with an "S." FiveAM, Stefil, LIFT, lisp-unit, RT, XLunit and Clickcheck are all packaged as ASDF systems. NST provides both this and a further level of ASDF integration with an extension of the class of ASDF systems to facilitate specifying how a system's test-op test operation should invoke NST.

SLIME [28] provides a richer Lisp development environment within Emacs. Tools can provide binding to streamline their use under SLIME. Although a number of systems list a SLIME front-end as a future goal, only Stefil documents its use under SLIME.

Updates.

The *Last* column describes the date of the last release, update or patch to the system. NST, FiveAM, LIFT, tester, and perhaps also Stefil seem to be under active development, or at least to be issuing occasional patches. We were unable to determine a last-release date for lisp-unit and MSL.TEST.

Other test systems.

We have omitted some Lisp test systems from the table: cl-smoketest [27] is for checking compilation of ASDF systems. FReT [20], as we mentioned above, seems still to be under initial development. EnclineUnit [8, 24], aka EnclineTest, seems to have been abandoned several years ago after its early releases; as of this writing its download links are non-functioning and we were unable to find an installable version.

[8]For example, *www.cs.chalmers.se/~rjmh/QuickCheck* .

5. CONCLUSION

We have introduced NST as a flexible and expressive unit testing framework for Common Lisp. While our survey of available Lisp test frameworks suggests a number of additional useful features for NST, most notably SLIME integration and results visualization, our experience suggests that NST is already quite useful for both larger and smaller projects.

The number of available testing frameworks for Lisp is, in part, a reflection of the ease which Lisp's introspective and dynamic features, as well as its uniquely expressive and flexible object hierarchy CLOS, lend to the construction and customization of program analysis and execution tools. This facility and the continuing evolution of software engineering techniques suggest to us that neither NST nor any of the other systems we survey here will be the last word on a canonical Lisp testing framework! We are hopeful that our presentation of NST's design will provide helpful design points to future test system authors.

Acknowledgments.

I am grateful to Michael J. S. Pelican, several anonymous reviewers and Jill Maraist for their detailed comments on this paper. NST itself benefited from design feedback and bug/usability reports from Michael Atighetchi, Steven A. Harp, Michael J. S. Pelican and especially Robert P. Goldman. This material is based upon work supported by the Defense Advanced Research Projects Agency under Contract FA8650-06-C-7606. The views, opinions, and/or findings contained in this article/presentation are those of the author/presenter and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense. Distribution Statement A: Approved for Public Release, Distribution Unlimited.

6. REFERENCES

[1] F. A. Adrian. CLUnit web site. http://www.ancar.org/CLUnit.html.

[2] F. A. Adrian. Using CLUnit: An introduction and tutorial. Shipped with CLUnit distribution [1].

[3] D. Bacon. Clickcheck and Peckcheck web site. http://www.accesscom.com/~darius/software/clickcheck.html.

[4] M. Baringer. *FiveAM*. Online user manual, *http://common-lisp.net/project/bese/FiveAM.html* .

[5] D. Barlow, R. P. Goldman, F.-R. Rideau, and Contributors. *ASDF Manual*. http://common-lisp.net/project/asdf/asdf.html.

[6] K. Beck. Simple Smalltalk testing: With patterns. *Smalltalk Report*, 1994. Updated June 2001, *http://www.xprogramming.com/testfram.htm*.

[7] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In P. Wadler, editor, *Proceedings of the International Conference on Functional Programming*, Montreal, 2000.

[8] Development, building, testing, and documentation aids. CLiki, the Common Lisp wiki. http://www.cliki.net/development.

[9] XLUnit CLiki entry. http://www.cliki.net/xlunit.

[10] P. Dietz. The GCL ANSI Common Lisp test suite. In J. White, editor, *International Lisp Conference*, Stanford University, June 2005.

[11] P. Foley. MSL.TEST web site. http://users.actrix.co.nz/mycroft/test-doc.txt, 2003.

[12] Franz, Inc. *The Allegro CL test harness*.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[14] P. Gold. Common Lisp testing frameworks. on-line article, Mar. 2007. http://aperiodic.net/phil/archives/Geekery/notes-on-lisp-testing-frameworks.html.

[15] P. Hamill. *Unit Test Frameworks*. O'Reilly Media, Nov. 2004.

[16] G. King. LIFT user's guide. http://common-lisp.net/project/lift/user-guide.html.

[17] G. King. LIFT web site. http://common-lisp.net/project/lift/.

[18] G. King. LIFT — the LIsp Framework for Testing. In R. de Lacaze, editor, *International Lisp Conference*, New York, Oct. 2003.

[19] A. Lendvai, T. Borbély, and L. Mészáros. Stefil: Simple TEst Framework in Lisp. Online documentation http://common-lisp.net/project/stefil/index-old.shtml (marked as obsolete) or http://dwim.hu/project/hu.dwim.stefil (described as newer by the older site, may not be accessible).

[20] S. Mishra. FReT (framework for regression testing) web site. http://www.sfmishras.com/smishra/fret/.

[21] J. Newton. HEUTE (Hierarchical Extensible Unit Testing Environment for Common Lisp) web site. http://www.rdrop.com/ jimka/lisp/heute/heute.html.

[22] S. Pedrazzini. clos-unit web site. http://www.lme.die.supsi.ch/~pedrazz/clos-unit/.

[23] S. Pedrazzini. A CLOS implementation of the JUnit testing framework architecture. In R. de Lacaze, editor, *International Lisp Conference*, San Francisco, Oct. 2002.

[24] Y. A. Rashkovskii. Encline web site. http://common-lisp.net/project/encline/.

[25] C. Riesbeck. LispUnit web site. http://www.cs.northwestern.edu/academics/courses/325/readings/lisp-unit.html.

[26] RT (Common Lisp regression tester) web site. http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/code/testing/rt/0.html.

[27] A. Shields. cl-smoketest CLiki entry. http://www.cliki.net/cl-smoketest.

[28] SLIME (the superior Lisp interaction mode for Emacs) web site. http://common-lisp.net/project/slime/.

[29] P. Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In J.-P. Jouannaud, editor, *Proc. Conf. Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128, Nancy, France, Sept. 1985. Springer.

[30] R. C. Waters. Supporting the regression testing of Lisp programs. *ACM Lisp Pointers*, 4(2):47–53, June 1991.