Prolog Unit Tests

Jan Wielemaker
HCS,
University of Amsterdam
The Netherlands
E-mail: wielemak@science.uva.nl

January 27, 2008

Abstract

This document describes a Prolog unit-test framework. This framework was initially developed for SWI-Prolog. The current version also runs on SICStus Prolog, providing a portable testing framework. See section 9.1.

Contents

1 Introduction

There is really no excuse not to write tests!

Automatic testing of software during development is probably the most important Quality Assurance measure. Tests can validate the final system, which is nice for your users. However, most (Prolog) developers forget that it is not just a burden during development.

- Tests document how the code is supposed to be used.
- Tests can validate claims you make on the Prolog implementation. Writing a test makes the claim explicit.
- Tests avoid big applications saying 'No' after modifications. This saves time during development, and it saves *a lot* of time if you must return to the application a few years later or you must modify and debug someone else's application.

2 A Unit Test box

Tests are written in pure Prolog and enclosed within the directives `begin_tests/1`,2 and `end_tests/1`. They can be embedded inside a normal source module, or be placed in a separate test-file that loads the files to be tested. Code inside a test box is normal Prolog code. The entry points are defined by rules using the head `test`(*Name*) or `test`(*Name, Options*), where *Name* is a ground term and *Options* is a list describing additional properties of the test. Here is a very simple example:

```
:- begin_tests(lists).
:- use_module(library(lists)).

test(reverse) :-
        reverse([a,b], [b,a]).

:- end_tests(lists).
```

The optional second argument of the test-head defines additional processing options. Defined options are:

blocked(+*Reason:atom***)**
The test is currently disabled. Tests are flagged as blocked if they are known not to work, but this is accepted as something that must be fixed in the future.

condition(*:Goal***)**
Pre-condition for running the test. If the condition fails the test is skipped. The condition can be used as an alternative to the `setup` option. The only difference is that failure of a condition skips the test and is considered an error when using the `setup` option.

setup(*:Goal***)**
Goal is run before the test-body. Typically used together with the `cleanup` option to create and destroy the required execution environment.

cleanup(*:Goal*)

Goal is always called after completion of the test-body, regardless of whether it fails, succeeds or throws an exception. This option or `call_cleanup/2` must be used by tests that require side-effects that must be reverted after the test completes. *Goal* may share variables with the test body.

```
create_file(Tmp) :-
        tmp_file(plunit, Tmp),
        open(Tmp, write, Out),
        write(Out, 'hello(World).\n'),
        close(Out).

test(read, [ setup(create_file(Tmp)),
             cleanup(delete_file(Tmp))
           ]) :-
        read_file_to_terms(Tmp, Terms, []),
        Term = hello(_).
```

true(*AnswerTerm Cmp Value*)

Body must succeed deterministically. AnswerTerm is compared to Value using the comparison operator *Cmp*. *Cmp* is typically one of =/2, ==/2, =:=/2 or =@=/2,[1] but any test can be used. This is the same as inserting the test at the end of the conjunction, but it allows the test engine to distinguish between failure of `copy_term/2` and producing the wrong value. Multiple variables must be combined in an arbitrary compound term. E.g. `A1-A2 == v1-v2`

```
test(copy, [ true(Copy =@= hello(X,X))
           ]) :-
        copy_term(hello(Y,Y), Copy).
```

AnswerTerm Cmp Value(*E*)

quivalent to `true`(*AnswerTerm Cmp Value*) if *Cmp* is one of the comparison operators given above.

fail

Body must fail.

throws(*Error*)

Body must throw *Error*. The error is verified using `subsumes_chk`(*Error, Generated*). I.e. the generated error must be more specific than the specified *Error*.

error(*Error*)

Body must throw `error`(*Error, _Context*). See `throws` for details.

all(*AnswerTerm Cmp Instances*)

Similar to `true`(*AnswerTerm Cmp Values*), but used for non-deterministic predicates. Each element is compared using *Cmp*. Order matters. For example:

[1]The =@= predicate (denoted *structural equivalence*) is the same as `variant/2` in SICStus.

```
test(or, all(X == [1,2])) :-
        ( X = 1 ; X = 2 ).
```

set(*AnswerTerm Cmp Instances***)**
: Similar to `all`(*AnswerTerm Cmp Instances*), but ignores order and duplicates with respect to *Cmp* in the comparison. Each element is compared using *Cmp*.

nondet
: If this keyword appears in the option list, non-deterministic success of the body is not considered an error.

sto(*Terms***)**
: Declares that executing body is subject to occurs-check (STO). The test is executed with *Terms*. *Terms* is either `rational_trees` or `finite_trees`. STO programs are not portable between different kinds of terms. Only programs *not* subject to occurs-check (NSTO) are portable[2]. Fortunately, most practical programs are NSTO. Writing tests that are STO is still useful to ensure the robustness of a predicate. In case sto4 and sto5 below, an infinite list (a rational tree) is created prior to calling the actual predicate. Ideally, such cases produce a type error or fail silently. See also the Prolog flag `occurs_check`.

```
test(sto1, [sto(rational_trees)]) :-
        X=s(X).
test(sto2, [sto(finite_trees),fail]) :-
        X=s(X).
test(sto3, [sto(rational_trees), fail]) :-
        X=s(X), fail.
test(sto4, [sto(rational_trees),error(type_error(list,L))]) :-
        L = [_|L], length(L,_).
test(sto5, [sto(rational_trees),fail]) :-
        L = [_|L], length(L,3).
```

Programs that depend on STO cases tend to be inefficient, even incorrect, are hard to understand and debug, and terminate poorly. It is therefore advisable to avoid STO programs whenever possible.

2.1 Test Unit options

begin_tests(*+Name***)**
: Start named test-unit. Same as `begin_tests(Name, [])`.

begin_tests(*+Name, +Options***)**
: Start named test-unit with options. Options provide conditional processing, setup and cleanup similar to individual tests (second argument of `test/2` rules).

Defined options are:

[2] See 7.3.3 of ISO/IEC 13211-1 PROLOG: Part 1 - General Core, for a detailed discussion of STO and NSTO

blocked(+*Reason*)
Test-unit has been blocked for the given *Reason*.

condition(*:Goal*)
Executed before executing any of the tests. If *Goal* fails, the test of this unit is skipped.

setup(*:Goal*)
Executed before executing any of the tests.

cleanup(*:Goal*)
Executed after completion of all tests in the unit.

2.2 Writing the test body

The test-body is ordinary Prolog code. Without any options, the body must be designed to succeed *deterministically*. Any other result is considered a failure. One of the options `fail`, `true`, `throws`, `all` or `set` can be used to specify a different expected result. See section 2 for details. In this section we illustrate typical test-scenarios by testing SWI-Prolog built-in and library predicates.

2.2.1 Testing deterministic predicates

Deterministic predicates are predicates that must succeed exactly once and, for well behaved predicates, leave no choicepoints. Typically they have zero or more input- and zero or more output arguments. The test goal supplies proper values for the input arguments and verifies the output arguments. Verification can use test-options or be explicit in the body. The tests in the example below are equivalent.

```
test(add) :-
        A is 1 + 2,
        A =:= 3.

test(add, [true(A =:= 3)]) :-
        A is 1 + 2.
```

The test engine verifies that the test-body does not leave a choicepoint. We illustrate that using the test below:

```
test(member) :-
        member(b, [a,b,c]).
```

Although this test succeeds, `member/2` leaves a choicepoint which is reported by the test subsystem. To make the test silent, use one of the alternatives below.

```
test(member) :-
        member(b, [a,b,c]), !.

test(member, [nondet]) :-
        member(b, [a,b,c]).
```

2.2.2 Testing semi-deterministic predicates

Semi-deterministic predicates are predicates that either fail or succeed exactly once and, for well behaved predicates, leave no choicepoints. Testing such predicates is the same as testing deterministic predicates. Negative tests must be specified using the option `fail` or by negating the body using `\+/1`.

```
test(is_set) :-
        \+ is_set([a,a]).

test(is_set, [fail]) :-
        is_set([a,a]).
```

2.2.3 Testing non-deterministic predicates

Non-deterministic predicates succeed zero or more times. Their results are tested either using `findall/3` or `setof/3` followed by a value-check or using the `all` or `set` options. The following are equivalent tests:

```
test(member) :-
        findall(X, member(X, [a,b,c]), Xs),
        Xs == [a,b,c].

test(member, all(X == [a,b,c])) :-
        member(X, [a,b,c]).
```

2.2.4 Testing error conditions

Error-conditions are tested using the option `throws(`*Error*`)` or by wrapping the test in a `catch/3`. The following tests are equivalent:

```
test(div0) :-
     catch(A is 1/0, error(E, _), true),
     E =@= evaluation_error(zero_divisor).

test(div0, [error(evaluation_error(zero_divisor))]) :-
     A is 1/0.
```

3 Using separate test files

Test-units can be embedded in normal Prolog source-files. Alternatively, tests for a source-file can be placed in another file alongside the file to be tested. Test files use the extension `.plt`. The predicate `load_test_files/1` can load all files that are related to source-files loaded into the current project.

4 Running the test-suite

At any time, the tests can be executed by loading the program and running `run_tests/0` or run_tests(+Unit).

run_tests
Run all test-units.

run_tests(+*Spec*)
Run only the specified tests. *Spec* can be a list to run multiple tests. A single specification is either the name of a test unit or a term ⟨*Unit*⟩:⟨*Tests*⟩, running only the specified test. ⟨*Tests*⟩ is either the name of a test or a list of names. Running particular tests is particularly useful for tracing a test:[3]

```
?- gtrace, run_tests(lists:member).
```

5 Tests and production systems

Most applications do not want the test-suite to end up in the final application. There are several ways to achieve this. One is to place all tests in separate files and not to load the tests when creating the production environment. Alternatively, use the directive below before loading the application.

```
:- set_test_options([load(never)]).
```

6 Controlling the test suite

set_test_options(+*Options*)
Defined options are:

load(+*Load*)
Determines whether or not tests are loaded. When `never`, everything between `begin_tests/1` and `end_tests/1` is simply ignored. When `always`, tests are always loaded. Finally, when using the default value `normal`, tests are loaded if the code is not compiled with optimisation turned on.

run(+*Run*)
Specifies when tests are ran. Using `manual`, tests can only be run using `run_tests/0` or `run_tests/1`. Using `make`, tests will be run for reloaded files, but not for files loaded the first time. Using `make(all)` `make/0` will run all test-suites, not only those that belong to files that are reloaded.

silent(+*Bool*)
When `true` (default is `false`), send informational messages using the 'silent' level. In practice this means there is no output except for errors.

load_test_files(+*Options*)
Load `.plt` test-files that belong to the currently loaded sources.

[3]Unfortunately the body of the test is called through meta-calling, so it cannot be traced. The called user-code can be traced normally though.

7 Auto-generating tests

Prolog is an interactive environment. Where users of non-interactive systems tend to write tests as code, Prolog developers tend to run queries interactively during development. This interactive testing is generally faster, but the disadvantage is that the tests are lost at the end of the session. The test-wizard tries to combine the advantages. It collects toplevel queries and saves them to a specified file. Later, it extracts these queries from the file and locates the predicates that are tested by the queries. It runs the query and creates a test clause from the query.

Auto-generating test cases is experimentally supported through the library `test_wizard`. We briefly introduce the functionality using examples. First step is to log the queries into a file. This is accomplished with the commands below. `Queries.pl` is the name in which to store all queries. The user can choose any filename for this purpose. Multiple Prolog instances can share the same name, as data is appended to this file and write is properly locked to avoid file corruption.

```
:- use_module(library(test_wizard)).
:- set_prolog_flag(log_query_file, 'Queries.pl').
```

Next, we will illustrate using the library by testing the predicates from library `lists`. To generate test cases we just make calls on the terminal. Note that all queries are recorded and the system will select the appropriate ones when generating the test unit for a particular module.

```
?- member(b, [a,b]).
Yes
?- reverse([a,b], [b|A]).
A = [a] ;
No
```

Now we can generate the test-cases for the module list using `make_tests/3`:

```
?- make_tests(lists, 'Queries.pl', current_output).
:- begin_tests(lists).

test(member, [nondet]) :-
        member(b, [a, b]).
test(reverse, [true(A==[a])]) :-
        reverse([a, b], [b|A]).

:- end_tests(lists).
```

8 Coverage analysis

An important aspect of tests is to know which parts of program is used (*covered*) by the tests. An experimental analysis is provided by the library `test_cover`.

show_coverage(*:Goal***)**
Run *Goal* and write a report on which percentage of the clauses in each file are used by the program and which percentage of the clauses always fail.

We illustrate this here using CHAT, a natural language question and answer application by David H.D. Warren and Fernando C.N. Pereira.

```
1 ?- show_coverage(test_chat).
Chat Natural Language Question Answering Test
...

==================================================================
                         Coverage by File
==================================================================
File                                       Clauses    %Cov %Fail
==================================================================
/staff/jan/lib/prolog/chat/xgrun.pl              5   100.0   0.0
/staff/jan/lib/prolog/chat/newg.pl             186    89.2  18.3
/staff/jan/lib/prolog/chat/clotab.pl            28    89.3   0.0
/staff/jan/lib/prolog/chat/newdic.pl           275    35.6   0.0
/staff/jan/lib/prolog/chat/slots.pl            128    74.2   1.6
/staff/jan/lib/prolog/chat/scopes.pl           132    70.5   3.0
/staff/jan/lib/prolog/chat/templa.pl            67    55.2   1.5
/staff/jan/lib/prolog/chat/qplan.pl            106    75.5   0.9
/staff/jan/lib/prolog/chat/talkr.pl             60    20.0   1.7
/staff/jan/lib/prolog/chat/ndtabl.pl            42    59.5   0.0
/staff/jan/lib/prolog/chat/aggreg.pl            47    48.9   2.1
/staff/jan/lib/prolog/chat/world0.pl           131    71.8   1.5
/staff/jan/lib/prolog/chat/rivers.pl            41   100.0   0.0
/staff/jan/lib/prolog/chat/cities.pl            76    43.4   0.0
/staff/jan/lib/prolog/chat/countr.pl           156   100.0   0.0
/staff/jan/lib/prolog/chat/contai.pl           334   100.0   0.0
/staff/jan/lib/prolog/chat/border.pl           857    98.6   0.0
/staff/jan/lib/prolog/chat/chattop.pl          139    43.9   0.7
==================================================================
```

Using `?- show_coverage(run_tests).`, this library currently only shows some rough quality measure for test-suite. Later versions should provide a report to the developer identifying which clauses are covered, not covered and always failed.

9 Portability of the test-suite

One of the reasons to have tests is to simplify migrating code between Prolog implementations. Unfortunately creating a portable test-suite implies a poor integration into the development environment. Luckily, the specification of the test-system proposed here can be ported quite easily to most Prolog systems sufficiently compatible to SWI-Prolog to consider porting your application. Most important is to have support for `term_expansion/2`.

In the current system, test units are compiled into sub-modules of the module in which they appear. Few Prolog systems allow for sub-modules and therefore ports may have to fall-back to inject the code in the surrounding module. This implies that support predicates used inside the test unit should not conflict with predicates of the module being tested.

9.1 PlUnit on SICStus

The directory of `plunit.pl` and `swi.pl` must be in the `library` search-path. With PLUNITDIR replaced accordingly, add the following into your `.sicstusrc` or `sicstus.ini`.

== :- set_prolog_flag(language, iso). library_directory('PLUNITDIR'). ==

The current version runs under SICStus 3. Open issues:

- Some messages are unformatted because SICStus 3 reports all ISO errors as instantiation errors.
- Only `plunit.pl`. Both coverage analysis and the test generation wizard currently require SWI-Prolog.
- The `load` option `normal` is the same as `always`. Use `set_test_options(load, never)` to avoid loading the test suites.
- The `run` option is not supported.
- Tests are loaded into the enclosing module instead of a separate test module. This means that predicates in the test module must not conflict with the enclosing module, nor with other test modules loaded into the same module.

10 Motivation of choices

Easy to understand and flexible

There are two approaches for testing. In one extreme the tests are written using declarations dealing with setup, cleanup, running and testing the result. In the other extreme a test is simply a Prolog goal that is supposed to succeed. We have chosen to allow for any mixture of these approaches. Written down as `test/1` we opt for the simple succeeding goal approach. Using options to the test the user can choose for a more declarative specification. The user can mix both approaches.

The body of the test appears at the position of a clause-body. This simplifies identification of the test body and ensures proper layout and colouring support from the editor without the need for explicit support of the unit test module. Only clauses of `test/1` and `test/2` may be marked as non-called in environments that perform cross-referencing.

Index