Unit Testing of COOLAPS Applications

Camilo Correa Restrepo, Jacques Robin[*]

December 17, 2021

Contents

1 Introduction

Experience has shown that, as with most engineered artifacts, systematically testing software is incredibly important in ensuring both its quality at a given instant and its ability to evolve without regressing as new features are added. To do this, several techniques have emerged in recent years to address these needs. Of particular importance to us is the notion of unit testing. This type of testing focuses on creating collections of tests for each individual "unit" within a particular application. Depending on the particular characteristics of one's programming environment, these "units" may be as high as the component level or may be as low as the individual functions (or in our case predicates) that build up the application.

Unit testing is a very powerful idea that allows one to:

[*]camilo.correa-restrepo@univ-paris1.fr, jacques.robin@univ-paris1.fr

1. test every single component of the application systematically and ensure that as the application evolves, the rest of the application remains stable;

2. document the intended use of a given "unit" in a more powerful and illustrative way than comments or other documentation can.

This makes it essentially indispensable in modern software development, and rarely will one ever see a commercial or production system that has not undergone rigorous unit testing. This, however, does not mean that these tests in themselves guarantee the quality of a software system: being programmed artifacts themselves, they are as prone to bugs as the original system, so care must be taken to not put blind faith in them. Furthermore, unit testing implies a considerable investment of time and effort that do not *directly* increase a given application capabilities whilst taking a comparable amount of code to produce. Another important question is the degree of coverage of the underlying system by the tests and, furthermore, the degree to which they indeed cover the possible cases that can arise for each "unit" (which may well be infinite).

With all that said, it remains a fantastically useful tool for engineering complex systems, and must be used whenever possible - to the furthest degree possible.

2 Tooling and Preparation

The underlying technologies of COOLAPS applications, notably Logtalk and SWI-Prolog, both include fairly robust testing libraries with almost all modern features one would expect from a testing framework. We will make use of Logtalk's "lgtunit" testing library, which, thankfully, requires no installation or special configuration on our part to use (assuming one has duly installed Logtalk).

To enable unit testing in a Logtalk environment, one needs, at a minimum the creation of two files. The first of these is a so-called tester file that will load all required libraries and enable necessary flags and then will invoke your test suites. It's structure is very similar to a Logtalk loader file, with exception that the final invocation of the initialization procedure is the **run** message on your loaded test files.

We will be testing our Wumpus world simulator available here: `https://github.com/kaiser185/wumpus_simul_ai4eu`. We will not go into details here as to how it functions: to make sense of all that is to follow it is

REQUIRED to at the very least familiarize oneself with the README and the basic idea of the Wumpus world.

The basic structure of our tester file is as follows (with explanatory comments):

```
:- initialization((
    %These first two lines enables verbose compilation
    %and load the testing library itself
    set_logtalk_flag(report, warnings),
    logtalk_load(lgtunit(loader)),
    %This line loads our application with debugging information
    logtalk_load(loader, [debug(on)]),
    %This line loads our test suite specification, ensuring it is treated as such
    logtalk_load(tests, [hook(lgtunit)]),
    %This line automatically executes our test suite after all compilation is done
    tests::run
)).
```

This is essentially all that is needed to begin performing tests. Executing this is as simple as running the following command:

```
$ swilgt tester.lgt
```

3 Writing the test suite

A Logtalk test suite is nothing more than an object that extends the **lgtunit** base object exposed by the library with the same name. While Logtalk includes several test dialects to define unit tests, for our purposes the simplest suffices. The basic structure of a test suite object is written as follows:

```
:- object(tests, extends(lgtunit)).

    %This is the actual unit test
    %It calls the goal(s) in the body of the predicate
    %and checks that some_predicate/1 succeeds in order to consider
    %that the test has been passed.
    test(my_test_id) :- some_predicate('some input').

:- end_object.
```

Now, let's examine a specific unit we would like to test within the application. In this case, we are going to examine the `breeze_percept/3` predicate within the `src/wwpercept.lgt` file. The code for this predicate is as follows:

```
:- object(wwpercept).

    % <Dependencies and other stuff>
    :- private(breeze_percept/3).
    breeze_percept(CurrentState, Percepts0, [breeze|Percepts0]) :-
        CurrentState::holds(at(agent,Position)),
        adjacent(CurrentState)::adjacent(Position, AdjacentPlaces),
        member(pit(_,_), AdjacentPlaces).

    breeze_percept(CurrentState, Percepts, Percepts) :-
        CurrentState::holds(at(agent,Position)),
        \+ (adjacent(CurrentState)::adjacent(Position, AdjacentPlaces),
        member(pit(_,_), AdjacentPlaces)).
:- end_object.
```

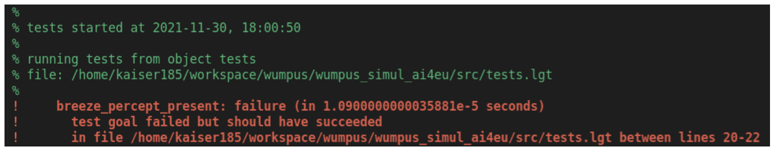
Admittedly, this is a rather complex predicate whose use and meaning is perhaps not immediately obvious, and whose comments I've omitted for reasons that will become clear further on. Before we dive deeper into it, let's examine two test cases written for it:

```
:- object(tests, extends(lgtunit)).
    % <Dependencies and other stuff>

    test(breeze_percept_present) :-
        user::tiny_pit(S),
        wwpercept<<breeze_percept(S, [], [breeze]).

    test(breeze_percept_not_present) :-
        user::tiny_no_pit(S),
        wwpercept<<breeze_percept(S, [], []).

    % <Other tests>
:- end_object.
```

With both the tests and the code in view, it becomes far easier to analyze what the meaning of the arguments is and what the intended use of the predicate is. If we observe the first test, we see that the first line of the body calls the `tiny_pit/1` predicate with some variable **S** as the argument. Considering both the code and the tests, it is clear that this corresponds to the `CurrentState` variable in the original code, which tells us that this goal is most likely a setup call. Indeed, it is; in fact, it simply unifies the variable with a predefined state (defined elsewhere) whose characteristics are known so the tests can be fully controlled, and the behavior easily predicted. If we look at what the test tells us, `breeze_percept/3` describes a relation between a given state and two lists of percepts. The crucial observation is that if the state so determines it, the second list will contain a `breeze` atom, and not otherwise. This begins to tell us far more about the predicate, than merely observing the code itself. One could even argue that the implementation of the predicate is itself **irrelevant** and that the crucial aspect is the interface it exposes, which is precisely what the unit tests document. The fundamental idea behind this is that tests aren't only tests, they are also documentation.

4 Finding bugs

Let's imagine we've made a simple mistake in our definition of `breeze_percept/3` and instead of adding the **breeze** atom, we are instead adding the **breze** atom to the list. While this is clearly a simple bug, it could break other parts of our application that depend on this particular predicate. If we run our test suite we would then observe the following:



```
%
% tests started at 2021-11-30, 18:00:50
%
% running tests from object tests
% file: /home/kaiser185/workspace/wumpus/wumpus_simul_ai4eu/src/tests.lgt
%
!     breeze_percept_present: failure (in 1.0900000000035881e-5 seconds)
!       test goal failed but should have succeeded
!       in file /home/kaiser185/workspace/wumpus/wumpus_simul_ai4eu/src/tests.lgt between lines 20-22
```

This would immediately indicate to us that something has indeed gone wrong with our test, and we would rather quickly discover our typo and correct our predicate definition. The one, rather large, drawback to our test framework as we are currently using it is that the information it can give as to **WHY** the test failed is limited. This can be combated to some degree through setting more the more complex `test/2` and `test/3` test dialects, in particular by using assertions in addition to seeing the test succeed (or fail if that's the expected outcome). The **test/2** and **test/3** dialects have the following structure:

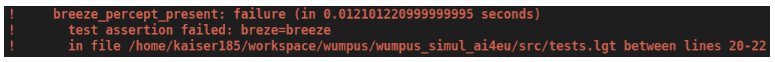
```
test(Name, Outcome) :- Goal.
test(Name, Outcome, Options) :- Goal.
```

Please see `https://logtalk.org/manuals/devtools/lgtunit.html` for details on the details and possible values for these arguments; for our purposes it suffices to say that (a) `Name` is the test identifier, (b) `Outcome` being `true` tells the test library that the test must succeed, and (c) `Options` is a list with which we can parameterize the test (the `setup(Goal)` option simply is a goal that must be used for setting up the test and can share variables with the actual test body). A word of warning, however, is in order before we continue: a test should be as simple as possible (besides the necessary setup conditions) because otherwise the cause of failures become very hard to determine.

To illustrate what being explicit about the expectation of seeing breeze would look like, we could rewrite the test as follows:

```
test(breeze_percept_present, true(( BreezeAtom = breeze ))) :-
    user::tiny_pit(S),
    wwpercept<<breeze_percept(S, [], [BreezeAtom]).
```

The expression `true(( BreezeAtom = breeze ))` within the test predicate tells the test framework that you expect the test to succeed and that the goal within the parenthesis must succeed **after** the test body has succeeded. This would give us the following output:



```
!     breeze_percept_present: failure (in 0.012101220999999995 seconds)
!       test assertion failed: breze=breeze
!       in file /home/kaiser185/workspace/wumpus/wumpus_simul_ai4eu/src/tests.lgt between lines 20-22
```

Notice that here there is an **EXTREMELY** important difference between these two test results: the former fails because the predicate call to `breeze_percept/3` failed due to having a different result in the third argument, whilst the latter does not have such a failure, it is the assertion itself which failed. One must tread quite carefully to avoid confusing these two, since they mean **very** different things.

5 QuickCheck

QuickCheck is an automatic test case generation method originally developed by the functional programming community. It has been subsequently been ported over to the Logtalk programming language and is an integral part of the **lgtunit** library. The fundamental idea behind it is that writing unit tests to cover large amounts of possible cases for given function (or predicate in our case) is rather laborious in the best of circumstances. To combat this, it is possible to automatically generate large amount of individual test instances with a single test definition.

A couple things must be noted about QuickCheck. The first of these is that it is not a replacement for unit tests, rather a complement to them. This is due to the fact that QuickCheck only works if you can identify a property to test that should hold across your entire range of test values and not for a single isolated instance. Thus, QuickCheck is an instance of what is termed **property-based** testing. The second is that determining these properties is not necessarily straightforward, and requires care to do correctly. A third and final note is that the values QuickCheck generates are random in nature, and care must be taken to avoid running into the pitfalls that random generation of instances can present.

With that said, let us first examine the predicate we are going to test with QuickCheck within our application:

```
:- category(sflux).

    %%More predicates to test
    :- public(holds/2).
    holds(F, [F|_]).
    holds(F, Z) :- Z=[F1|Z1], F\==F1, holds(F, Z1).
    %%More predicates

:- end_category.
```

Now, this predicate, in essence, does something very simple, it recursively checks for the presence of an element in a list. Its semantics (and origin) are within the Special Flux Kernel (and thus Special Fluent Calculus) described by Michael Thielscher in his book Reasoning Robots (2005) to which readers are referred for details. What's important to our discussion here is that we want to be able to test some property that holds across large amounts of instances **and** that is independent of the implementation details. We, of course, are aware that the predicate looks recursively through the list to do so, but the points to be outlined in this section apply far more generally.

Logtalk's QuickCheck implementation relies very heavily on the **type** system withing Logtalk, specifically the user-extensible **type** (`https://logtalk.org/manuals/libraries/types.html`) and **arbitrary** (`https://logtalk.org/manuals/libraries/arbitrary.html`) libraries that respectively implement functionality to perform type checking and and generate arbitrary terms of a given type. Since we will be testing a predicate whose semantics indicate to us that it is defined for fluents and states (represented as lists of fluents), we must create our own definitions of what a **fluent type** is and represents. To do this, we must **extend** the definitions of the `type::type/1` and `type::check/2` predicates. We will create a dedicated object where we will define both the types and where, further on, we will define the arbitrary term generation that we will use to run our QuickCheck tests. To begin with, let's define the object with our `fluent` type:

```
:- object(wwtypes).

    :- use_module(lists, [member/2]).

    :- multifile(type::type/1).
    type::type(fluent).
    :- multifile(type::check/2).
    type::check(fluent, Fluent) :-
        type::check(term, Fluent),
        functor(Fluent, Functor, _),
        member(Functor,[at, out, carries, alive, dead]).

:- end_object.
```

As it can be seen, we are defining fluents, within our application, to essentially be valid prolog terms (the call to `type::check(term,Fluent)` is in essence asking the system to determine that the `Fluent` variable is indeed a term as defined by the backend Prolog interpreter) whose functors are within those defined for our Wumpus world case. The `type::type/1` predicate we define is simply the declaration that `fluent` is one of the types that can be used by the Logtalk type system. These predicates will be used indirectly by the QuickCheck implemenation to check that the generated types correspond to our definition.

Now, with our `fluent` type defined, we must be able to **generate** arbitrary terms that conform to our type specification. To do so, we must extend the `arbitrary` category's `arbitrary::arbitray/1` and `arbitrary::arbitrary/2` predicates that will allow us to both define the types that can be generated arbitrarily and the actual generation of terms of that type. In so doing, we will arrive at the following version of our earlier object:

```
:- object(wwtypes, imports(functor_handling)).
```

```
    :- use_module(lists, [member/2]).

    :- multifile(type::type/1).
    type::type(fluent).
    :- multifile(type::check/2).
    type::check(fluent, Fluent) :-
        type::check(term, Fluent),
        functor(Fluent, Functor, _),
        member(Functor,[at, out, carries, alive, dead]).

    :- multifile(arbitrary::arbitrary/1).
    arbitrary::arbitrary(fluent).
    :- multifile(arbitrary::arbitrary/2).
    arbitrary::arbitrary(fluent, Arbitrary) :-
        random::member(Functor, [at, out, carries, alive, dead]),
        ::handle_functor(Functor, Arbitrary).

:- end_object.
```

We have also augmented this code with a category that defines how the terms are created depending on the functor at hand, as follows (though it must be noted we only show the first of the clauses for the `handle_functor` predicate for the sake of brevity; all the others are much the same):

```
:- category(functor_handling).

    :- private(handle_functor/2).
    handle_functor(at, at(Entity, Place)) :-
        random::member(EntityBase, [agent, wumpus]),
        random::member(PlaceBase, [exit, wwplace, pit]),
        type::arbitrary(list(positive_integer,3),[X,Y,Id]),
        Entity=..[EntityBase, Id],
        Place=..[PlaceBase, X, Y].

    %Other clauses for the other possible functors%
:- end_category.
```

With these definitions in place, it is now finally possible to begin defining our QuickCheck tests. To begin with, we must identify a property that must hold for the `holds/2` predicate. The simplest property that is invariant to the amount of fluents in the state is the fact that every single member of the list representing the state must **hold**. The triviality of this property notwithstanding, it faithfully reflects the intended semantics of the `holds/2` predicate, and, moreover, is completely independent of the implementation, that while known *a priori*, should not be a relevant factor for the property itself. We can express such a property with following predicate (note that we `import` the `sflux` category where the `holds/2` predicate is defined into our test suite object to be able to call the predicate directly without an intermediary object):

```
holds_prop(Fluents) :-
    forall(member(Fluent, Fluents),::holds(Fluent, Fluents)).
```

With this property duly defined, all that remains is to include within our test suite a QuickCheck test definition. This is accomplished through the use of the `quick_check/2` and `quick_check/3` test dialects. These test definitions have the following form:

```
quick_check(Test, Template, Options).
quick_check(Test, Template).
```

The `Test` and `Options` variables are much the same as they were for the `test/2-3` definitions above, with some further specific options of QuickCheck tests. However, we will only require the simpler of the two, namely `quick_check/2` for the purposes of this tutorial. The `Template` is a predicate template as described in `https://logtalk.org/manuals/devtools/lgtunit.html?highlight=template#quickcheck`, where one utilizes mode declarations to control which terms are to be generated automatically as "input" to the property predicate. Putting it all together, we will add the following to our test suite:

```
quick_check(holds_prop, holds_prop(+non_empty_list(fluent))).
```

Here we make use of the parametric type `non_empty_list` from the `arbitrary` library mentioned above, which in turn internally uses our type definition to generate the list of fluents we need to test our desired property. This is, then, the basic operation of the QuickCheck.

6 Conclusion

We have seen that unit tests within COOLAPS applications are a very powerful tool to both ensure the quality of the individual units within our application and to ensure that our application is stable as it evolves. It is an indispensable part of software development, and should accompany all steps of the development process. It is our hope that this tutorial will prove valuable in testing COOLAPS applications moving forward.