

A Tool to generate unit tests

UNICA

September 2, 2009

Abstract

The agent-based representation of the economic model is made by a lot of functions that we need to be sure that it are working properly. The smallest function for an agent could be considered the transition function, this function allows the agent to act and to communicate with the others agents. The correct operation of all transition functions is a necessary condition for the correct functioning of the entire system. The best way to test small parts of the system is the unitary test provided with a tool that collect all units in one o more suits and able to run automatically. We present a tool that provide an automatic way to write and run a collection of suites generating code from a collection of a suites, where each suite is described by a specified markup language.

Contents

1	Introduction	1
1.1	The unit-test generator tool:basics ideas	1
2	General description	2
2.1	The configuration file	3
2.2	The suite	3
2.2.1	constants	3
2.3	The unit test	4
2.3.1	The declarative part	4
2.3.2	The fixture	5
2.3.3	The expected values	5
3	Quick start guide	6
3.1	Downloading	6
3.2	Making	6
3.3	Generating test case and suite organization	6
3.4	Running	6
3.5	The examples: rules.xml	7

1 Introduction

1.1 The unit-test generator tool:basics ideas

The agent-based representation of the economic model is made by a lot of functions that we need to be sure that it are working properly. The smallest function

for an agent could be considered the transition function, this function allows to the agent acting and communicating with the others agents. The correct operation of all transition functions is a necessary condition for the correct functioning of the entire system. The best way to test small parts of the system is the unitary test provided with a tool that collect all units in one or more suites and able to run automatically. In a system is large it is a good practice to divide the system in modules, which have to be tested with an automatic tool respecting the modular organization. In order to do this specific ...

The rules that is following satisfy the lexical-grammar definition that we have defined before:

2 General description

The automatic tool to run unitary test run a collection of suites, where each suite contain a collection of unitary tests. Each suite is related basically to a module maintaining the preexistent subdivision, but it is possible to build more suites for a single module, for example we can write three suites for the financial market regarding each agents that act in the financial market, so we will have a better subdivision. The set of the suites is collected in a xml file, this file allows to configure the system and group the suites.

2.1 The configuration file

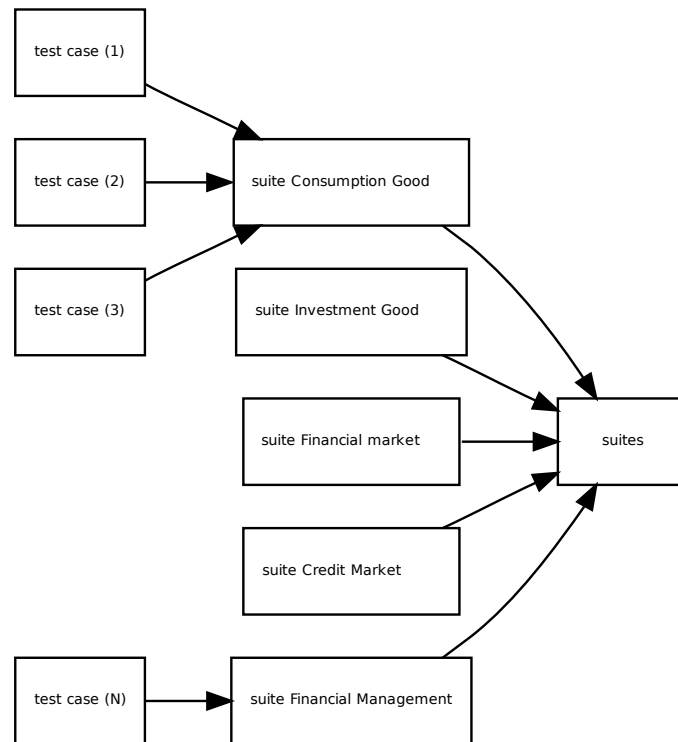
The set of the suites of the entire model are grouped by a configuration file, this file are used to configure the paths of each suite and the path of the model. This file is placed in the top of model in the same folder that contains the model description. The previous code describes the configuration of the testing for the eurance model, it contains two fundamental tags that are following:

- the tag **modelPath**, associate the system of test to the FLAME agent model;
- the tag **suitePath**, it allows to add a new suite associating its path.

2.2 The suite

The suite is described by a file, this file has the extension .xml and has to be placed in the linked module, in the same folder of the associated sub-module. The suite is identified by a name and is characterized by a collection of unitary tests. The agent model is set with a list of constants that need to be specified. The composition and elements of a suite is opened with the tag called **suite** and its specification is characterized by the following tags:

- the tag **name**: this tag specified the suite name, this name must be unique in the system.
- the tag **constants**, this tag contain the values of each constants. We must specify only the constants that are used in the collection of test-cases, this definition is explained with more details in the next paragraph.
- the tag **unittest**: this tag open the unit-test specification and it could be written more times , each time that we have to define a test-case.



```

<suites>
<modelPath>eurace_model.xml</modelPath>
<suitePath>FINANCIAL_UG/suite.xml</suitePath>
<suitePath>Government_GREQAM/suite.xml</suitePath>
<suitePath>Cons_Goods_UNIBI/suite.xml</suitePath>
</suites>

```

Figure 1: the configuration file suites.xml

```

<constants>
<firm_planning_horizon>10</firm_planning_horizon>
<seed>10000</seed>
</constants>

```

Figure 2: the constants definition

2.2.1 constants

The tag constant permit to open a tree that speciefies the values of the constants which are defined in the related agent model. Moreover, there are a set of contants that can be defined only in the system of test, as an example the the *seed* and the *current_day*. The *seed* is a particular and fondamental constants useful to set the seed of the random number generator. The *current_day* is the present day of a calendar, this constants is useful because more functions could change depending of the day of the calendar. The constants are defined as the example figure 2.2.1. The example show the value association of two constants , the constants named *firm_planning_horizon* e the constants *seed*.

2.3 The unit test

A unit test is specified with the tag unittest and permit to define a test case. More test-case could be written for a particulare function, writing more fixture in order to fill the greater part of cases. A complete verification of a program or a simple part of a program at any stage in the life cycle can be obtained by performing the test process for every element of the domain. If each instance succeeds, the program is the program is verified; otherwise,an error has been found. This testing method is known as exhaustive testing and is the technique that will guarantee the validity of a program. Unfortunately, this technique is not practical. Frequently, functional domains are infinite, or even if finite, sufficiently large to make the number of required test instances infeasible. The test case is a good test In order to reduc The unit test is set by the following items:

- the **name of the test**. This name has to be unique in the system.
- the **name of the transition function** that will has to be tested.
- the name of the owner of the transition function. This part specifies the type of agent (for example Household or Firm) that are subject to the test.
- the **declarative part**. This part is important because it specifies how the assertion parts have to be built .
- the **fixture**.
- the **expected values**.

2.3.1 The declarative part

The declarative part specifies the variables and messages that are subject to assertions. This part is opened with the tag called *declaration* and contains a list of variables and a list of messages. The following code shows an example of a typical declarative part:

```
<declaration>
<variables>
  <variable>
    <type>Asset_array</type>
    <name>assetsowned</name>
  </variable>
  <variable>
    <type>Order_array</type>
    <name>pendingOrders</name>
  </variable>
</variables>
<messages>
<message>
  <name>order</name>
</message>
</messages>
</declaration>
```

The xml code above shows two variables and a messages that are subject to assertion. The **test_generator** tool generate three types of assertion: the first assertion is related to the *assetsowned*, the second to *pending_orders* and finally the third is related to the *order* message.

2.3.2 The fixture

The fixture is a set of memory variables and input messages, initialized to proper values, used as repeatable input data for the tests. Each time a test case is run, its fixture is reinitialized, because previous tests might have corrupted the fixture, making the test fail not due to errors in the code, but to wrong test data. By defining a fixture, you decide what you will and won't test for. A complete set of tests for a transition function will have many fixtures, each of which will be used by many tests, in many ways. This part is opened with the tag called **fixture** and contains a list of the initialization values of agent's variables and a list of the initialization values of messages. The following code shows an example of a typical fixture part:

```
<fixture>
<Household>
<id>1</id>
<assetsowned>{{1,10,100}}</assetsowned>
<pendingOrders>{{1,50,90,1}}</pendingOrders>
</Household>
<messages>
<order_status>{{1,1,110,1}}</order_status>
</messages>
```

```
</fixture>
```

2.3.3 The expected values

A Test Case stimulates a Fixture and checks for expected results. If the tests are unsuccessful, they have to provide helpful information about the kind of error and about his location, the system that launch the test shows a summary of all test belonging from all suite. To this purpose, the framework is endowed with standard checking functions (Checks) able to test Boolean conditions and to report automatically the results and the system state in the case of failure. The checks need an expected state of the agent and the expected output messages. The expected values part is opened with the tag called **expected_states** and contains a list of the expected values of agent's variables and a list of the expected values of output messages. The following code shows an example of a typical fixture part:

```
<expected_states>
<Household>
<id>1</id>
<assetsowned>{{1,10,100}}</assetsowned>
</Household>
</expected_states>
```

The xml code above shows two agent's variables that are the expected state of the Household. The expected.state parts don't contain a list of expected messages because the related transition function don't involve any output message.

3 Quick start guide

Here is a set of steps for setting up and generate C code of unit test and run the test automatically . Details and instructions for a more thorough tour of **test_generator** features, including installing, validating, and using the performance evaluation tools, are given in the following sections.

3.1 Downloading

The first step is to download the test_generator and install any necessary files. It needs the libxml and CUnit library that have to be installed properly. The way to get test_generator is to use the repository at the address <http://ccpforge.cse.rl.ac.uk/svn/eurace/tests>. Get the the entire folder unit-test-generator that contain the following files: main.code.c, Makefile , datatype.c.tmp , header.h.tmp , Suite.h.tmp, Suite.c.tmp, messages.c.tmp, LauncherTest.c.tmp Now you are ready to build.

3.2 Making

Before you can use **test_generator**, you must make it.

Make test_generator

```
% make
```

This may take a while, depending on the load on your system and on your file server, it may take anywhere from a few seconds to an minut or more.

Figure 3: the configuration file rules.xml

3.3 Generating test case and suite organization

The test_generator generate code by using the model information and the information contained in the files called suite.xml and present in each module. The code generated is compounded by some C code files and a Makefile called Makefiletest,

3.4 Running

Run a simple integrity test:

```
% cd ../integrity_tests
```

```
% ./integrity_tests/rules.xml target.txt
```

At this point you have run an integrity test on your system.

3.5 The examples: rules.xml

The Integrity distribution contains a simple programs, which are located in the integrity source tree. The rules.xml 3.5 is a simple example which is explained how configure an integrity test.