



Project no.
035086

Project acronym
EURACE

Project title

An Agent-Based software platform for European economic policy design with heterogeneous interacting agents: new insights from a bottom up approach to economic modelling and simulation

Instrument STREP

Thematic Priority IST FET PROACTIVE INITIATIVE “SIMULATING EMERGENT PROPERTIES IN COMPLEX SYSTEMS”

Deliverable reference number and title

D8.3: Development of agile practices and tools for generating and updating economic models

Due date of deliverable:
28/02/2009

Actual submission date:
27/03/2009

Start date of project: September 1st 2006

Duration: 39 months

Organisation name of lead contractor for this deliverable
Università degli Studi di Cagliari - UNICA

Revision 1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	X

Development of Agile practices and Tools for generating and updating Economic Models

Authors:

Sabrina Ecce, Mario Franco Locci, Michele Marchesi

University of Cagliari

Contents

1	Introduction	6
2	Software Life Cycle based on Agile Testing practices	8
2.1	Testing Process	10
2.1.1	Verification and Validation (V&V)	10
2.2	Unit Testing	11
2.2.1	Unit Testing concepts	12
2.2.2	Unit Testing in FLAME	13
2.3	Functional Testing	14
2.4	Automatic Testing Tools for EURACE	14
3	Automatic Unit-Test Generator Tool (AGT)	16
3.1	The configuration file	17
3.2	The Suite	18
3.2.1	Constants	18
3.3	The Unit Tests	18
3.3.1	The declarative part	19
3.3.2	The fixture	19
3.3.3	The expected values	21
4	Automatic Integrity Testing Tool (AIT)	22
4.1	Consistency rules of EURACE model	22
4.2	The grammar rules of AIT	23
4.3	Specifying integrity rules	24
4.4	Application output	25
A	Using Automatic Unit-Test Generator Tool	28
A.1	Set up of the tool	28
A.2	Test running	28
B	Using Automatic Integrity Testing Tool	30
B.1	Set up of the tool	30
B.2	Running integrity tests	30

List of Figures

1	Verification and Validation (V&V)	11
2	Modular organization of the test suites. There is a test suite for each sub-module of the system.	16
3	Modular organization of the test suites in the case of the Financial Market module. .	17
4	An example of configuration file. of the suites	17
5	An example of constants definition.	18
6	Declarative part of the tool input.	20
7	The definition of a Fixture.	20
8	An example of expected values of a test.	21
9	Some integrity rules written accordingly to lexical-grammar of AIT. rules	24
10	An example of integrity rule declaration.	25
11	An example results due to integrity rule application.	26
12	Inputs and outputs of the test_generator	28
13	Unit test console	29

Abstract

The EURACE project produced a large econometric, agent-based model, and a software environment to develop and run economic models. The environment includes many different components – at various granularity levels – together with methods, practices and tools to change and put at work the models. This means that EURACE system is subject to continuous changes.

Such complexity and change rate make impossible for any single person to control the system and its evolution completely. If changes to the system are made improperly, its complexity will quickly hinder and ultimately make impossible to reach the goal of having complex economic models producing results that can be trusted.

In this report, we present some practices and tools to manage the activity of making changes to EURACE system, based on the agile practices of automated testing. The aim of automated testing is not only to improve software quality, but also to shield the system from unwanted side-effects due to changes. In fact, relying on a well chosen set of automated tests can greatly improve the probability of finding errors and failures introduced while modifying the system.

After introducing the issues, practices and benefits of automated testing, in the context of EURACE system, this document provides a detailed description of two state-of-the-art, automated testing tools developed to cope with a large system, based on X-Agents.

The first tool, called "Automatic Unit-Test Generator Tool" (AGT), provides an automatic way to run automatically unit testing process in EURACE. With AGT, unit tests can be declared and written in a language based on xml, with no need to hard-code them directly in C language. AGT uses the open-source CUnit test suite, which provides a console to control test execution.

The second tool is called "Automatic Integrity Testing Tool" (AIT). It is helpful in the validation phase of EURACE model, being able to check integrity rules in the simulation outputs. Also in this case, the user needs only to write the integrity rules in xml language, and AIT takes care to verify them and to print the test results in an output file.

1 Introduction

The EURACE project produced a large econometric, agent-based model, that is implemented by a large software system. This system is more an environment to develop economic models rather than a single model. The environment includes many different components – at various granularity levels – together with methods, practices and tools to change and put at work the models. This means that EURACE system is subject to continuous changes.

In fact, EURACE system can be used in two possible ways. In the simpler usage, a specific model is produced, and it is used and stimulated varying its parameters – number and composition of agents, initial endowments, behavioral parameters, etc. A more complex usage entails to make changes to the model itself, for instance adding new behaviors related to investment strategies of households and firms, or changing the exploitation of technological innovation by the firms.

While the former kind of usage needs still some validation, because improper parameter setting can produce unrealistic results, as can happen with every model, the latter way entails much more problems. After devising the changes and updates in economic models, some software modification are made, with the potential to produce unwanted side-effects in other parts of the system. Moreover, an accurate verification must be made to check whether economic invariants, like account exact balancing, still hold throughout the model.

Such complexity and change rate make impossible for any single person to control the system and its evolution completely. If changes to the system are made improperly, its complexity will quickly hinder and ultimately make impossible to reach the goal of having complex economic models producing results that can be trusted.

In the final phase of the project, we need to devise a method and some practices and tools to manage the activity of making changes to EURACE system. This can be obtained using the agile practices of testing based on a automated test strategy. The aim of automated testing is not only to improve software quality, but also to shield the system from unwanted side-effects due to changes. Clearly, it is not possible to guarantee absolute software integrity without an exhaustive testing, which is out of question for a system of the complexity of EURACE. However, relying on a well chosen set of automated tests can greatly improve the probability of finding errors and failures introduced while modifying the system.

The advantage of having a suite of automated tests, to be executed frequently, whenever changes are made to the system, is that many errors are caught by tests, thus lowering the need of debugging sessions, which are often very long and resource-consuming. The obvious disadvantage is that designing and writing tests is costly. Some software engineering studies shown that, especially in large systems, the cost of writing tests is more than repaid by the reduction of the amount of time spent working with debugging tools. The need of frequently debugging is a sign that the unit tests are lacking coverage, or are trying to test too much functionality at once.

It is remarkable what Martin Fowler wrote in Chapter 4 of *"Refactoring"* book [3]:

If you look at how most programmers spend their time, you'll find that writing code is actually a small fraction. Some time is spent figuring out what ought to be going on, some time is spent designing, but most time is spent debugging. I'm sure every reader can remember long hours of debugging, often long into the night. Every programmer can tell a story of a bug that took a whole day (or more) to find. Fixing the bug is usually pretty quick, but finding it is a nightmare. And then when you do fix a bug, there's always a chance that another one will appear and that you might not even notice it until much later. Then you spend ages finding

that bug.

Manual debugging is usually a slow, and tedious process. This is especially true for a system complex and relying on so many layers and subsystems like EURACE. It is easy to spend many hours tracking down a single logic error. Making frequently debugging reduces productivity and makes development schedules much less predictable because a single manual debugging session could extend the time required to develop the software.

EURACE system is characterized also by a completely asynchronous flow of control, because the messages sent by agents to other agents are processed in a non-predictable order. Moreover, since EURACE economic models are quite complicated, its code tends to have many branches or loops, whose control parameters change at each time step. This is very difficult to debug, and characterizing exactly what causes the failure may require many attempts, and an average time even longer than in typical software systems.

For all these reasons, we concentrated on providing automated testing tools which, given the specific characteristic of EURACE project, are not addressed by standard automated testing frameworks, such as C-Unit. Using the presented tools, we hope to minimize the time spent in debugging. Overall, the strategies to create automatic tests we followed are introduced in Gerard Meszaros's book [5].

In this document, first we'll summarize key concepts for how to create unit tests on EURACE system; then we will describe an automatic unit testing tool (Automatic Unit-Test Generator Tool (AGT))able to run automatically unit testing process, and to control test execution. We will then describe another automatic testing framework (Automatic Integrity Testing Tool (AIT)), helpful to the validation phase of the EURACE model. AIT framework allow to analyze the consistency of the model results by checking the various values computed in various events of the simulation.

2 Software Life Cycle based on Agile Testing practices

As every other software development project, Eri framework follows its software life cycle, which is quite similar to a standard software life cycle. Performing all the following life cycle activities following general best practices and practices specific to EURACE is essential to produce, in a controlled way, a system of good quality:

- **Requirements Elicitation:** in this phase, the requirements necessary for the application are identified. Requirements can be *functional requirements*, that describe what the system is supposed to do. In the case of EURACE, the basic functional requirements are those included in the project contract. These requirements have been subsequently developed, and are mostly described in term of mathematical equations prescribing the behavior of EURACE sub-systems and agents. *Non-functional requirements* are no less important than functional ones, and express other properties the system must exhibit, like reliability, security, ease of use and efficiency). In the case of EURACE, the most important non-functional requirement is clearly efficiency, due to the dimension of the system to be modelled and simulated. Other obviously important non-functional requirements are ease of use and evolvability. Other requirements can be related to the development process, for instance prescribing a certified quality control process. In the case of EURACE, there are not externally imposed process requirements. EURACE functional requirements are expressed mainly in the project deliverables. Basic functional requirements describe the software framework to develop; they are described, for instance, in D1.1 (X-Agent framework and software environment for agent-based models in economics), D1.3 (Graphical user interface to build and develop complex agent-based models in economics), D1.4 (Porting of agent models to parallel computing). The functional requirements specifying the model are contained in deliverables D5.1 (Agent based models of good, labour and credit markets), D6.1 (Agent based models for financial markets), D7.1 (Agent based models for skill dynamics and innovation).
- **Analysis** is the phase where it is determined whether the stated requirements are unclear, incomplete, ambiguous, or contradictory, and then where these issues are resolved. In practice, it amounted essentially to make EURACE software developers knowledgeable about the economic models to build, and able to work together with economists. In this way, they were able to fully understand EURACE economic models, and to verify their coherence under a software engineering perspective. Analysis phase usually produces one or more analysis documents, also in the form of blueprints, that summarize requirements and are the basis to perform system design. In the case of EURACE, analysis documents are merged with design documents, consequently the last part of analysis phase, and design phase were performed together.
- **Design** determines the software system architecture, in terms of hardware configuration and software to use. In EURACE, this phase essentially produced the detailed architecture of software agents, to be implemented using the Flame framework. This architecture includes agents' definition, message definition, agent statecharts and interface definition. It is documented in deliverables D5.3 (Software module for the agent based models of good, labour and credit markets), D6.3 (Software module for the agent based models for financial markets), D7.3 (Software module for the agent based models for skill dynamics and innovation).

- **Coding** represents implementation of the system, including in the case of EURACE its parallelisation. The product of this phase is the working software system.
- **Testing** verifies the correctness of the developed modules by unit tests and functional tests. This phase should guarantee both Validation and Verification (V & V) of the system. Verification guarantees that the software implements a specific function correctly with respect to the requirements. Validation guarantees that the software is what the customer actually requires, and that it performs the functions that the customer needs in a way that is acceptable to the customer and to the end users.
- **Evolution** consists in system maintenance (corrective, adaptive and perfective).

A well organized life-cycle process, together with sound practices to perform all the activities listed above, is absolutely necessary to develop good quality software.

There are various kinds of software development processes, but when the requirements are not clearly stated and stable since the beginning of the project, the best approach to software development is the use of the so called "Agile" approach [2]. Agile processes are a set of processes, practices and tools for software development based on iterative and incremental development, with very short interactions, relying on self-organizing teams. They are able to manage requirements that are changing, even when coding is in an advanced stage.

In the case of EURACE, while basic requirements are stable and defined by the contract, the specific economic agent-based models and their relationships has been found during the project itself. Being there requirements derived by research activities, also taking advantage of the feedback gained by running earlier models, they were intrinsically changing. Hence the need of an agile process for development.

An important aspect of agile approaches is that they are based on code and testing, and not on analysis and design performed up-front. In particular, testing is of paramount importance in the whole development process of a software system, because it substantially contributes to ensuring that a software application does everything it is supposed to do, in particular after the many changes required to the code to follow changes in requirements. In practice, automated testing is and "assurance" against unintended side-effects due to changes in the code. Of course, in practical systems like EURACE, testing cannot be exhaustive, giving a 100% guarantee that the system is not broken by changes, but can give a good enough guarantee, especially compared with approaches with no testing, or no automated testing.

In an "Agile" software development process, during each development iteration, it is suggested to write tests even before writing the code. This is called "Test Driven Development" (TDD) [1], a practice where developers systematically write a test, and then write just enough production code to fulfill that test.

In EURACE we did not go so far, also because Flame framework and C language lack the flexibility of object-oriented languages TDD was devised for. However, the present guidelines for developing EURACE software are mostly about how to write tests of various kind for the software developed.

In the following section, we summarize briefly the objectives and the key concepts of testing practice. More details on this subject can be found in deliverable D1.2.

2.1 Testing Process

In accordance with the definition of the *IEEE*, software testing is the process of analyzing a software item to detect bugs and to evaluate its features by going through the phases of requirements elicitation, analysis, design, implementation, integration and release.

Generally speaking, the testing process has the following main two goals:

- Demonstrating to developers and to customers that the software satisfies all requirements. For this reason, there should be at least one test for each user requirement, and for each functionality of the software system that will be incorporated in the release. Some software systems have an explicit acceptance testing phase, during which the user formally verifies that the deployed system is compliant to the requested specifications. This validation test is performed running a set of test cases specified by the customer, that ensure the system is working correctly.
- Discovering the errors or the defects of the software, that is software behavior that is wrong or does not comply with its specifications. Defect testing aims to find all kinds of undesirable behaviors of the system, such as crashing, undesirable interaction with other systems, wrong computations.

In order to reach these goals, it's fundamental to execute the Verification and Validation (V&V) activities of the software through the testing practice.

2.1.1 Verification and Validation (V&V)

The verification and validation activities should be executed at each stage of the software development process. At first we test, or review, system requirements; then we review the system design, or architecture. During and after each coding session we also write and run tests on the developed code; eventually, we test and validate the final software system. The Verification and Validation processes are not just the same, in fact:

Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase (IEEE 1990). It guaranties that the software implements a specific function correctly with respect to the requirements documentation. It answers the question: "Are we building the product right?"

Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements (IEEE 1990). It guarantees that the software is what the customer requires and that it performs the functions that the customer needs in a way that is acceptable to the customer and to the end users. It answers the question: "Are we building the right product?"

Typically, the V&V phase can be covered executing a dynamic analysis of the system. In other words, we can execute an evaluation process of the software system or his components based on the observation of its run-time behaviour.

As shown in Figure 1, the verification process allows to check if the software implementation is consistent with the requirements specification, whereas the validation process allows to check if the implementation software is consistent with informal requirements, in other words, whether it satisfies the customer needs.

In the context of EURACE, verification is performed on the software modules (agents, functions, message board), to check if they produce results consistent with the mathematical

models they implement. With a proper framework, verification can be automated. Validation, on the other hand, checks if the overall behaviour of EURACE system produces sound economic results. This means basically that the statistical properties of economic series generated by the model must be consistent with their real behaviour. Validation can be automated only to a limited extent, and most of it must be performed by visual inspection of system output by an expert. However, since the outputs of EURACE system can be very many, their automated management through a proper user interface, or other software modules can help.

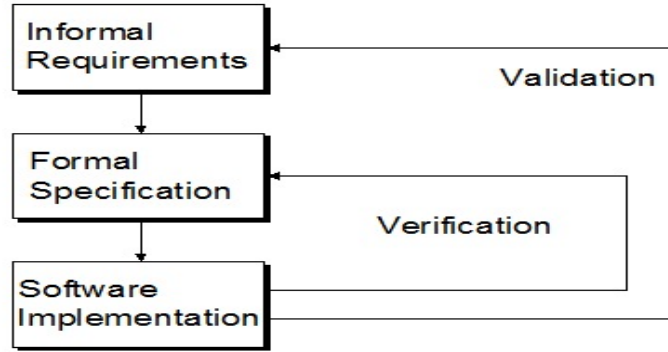


Figure 1: Verification and Validation (V&V)

Generally speaking, all dynamic testing techniques which we consider for V&V of the software, can be categorized in two kinds:

- WHITE BOX testing, which includes all structural tests. The goal is to verify all implementation details.
- BLACK BOX testing, which includes all functional tests. In this case, tests can verify only externally visible behaviour and are independent of the implementation inside the system under test. The BLACK BOX testing determines if a program does what it is supposed to do according to functional requirements.

There are distinct levels of testing, but in order to test the EURACE system, we have focused mainly on two levels of testing:

1. *Unit Testing*: it's a kind of "white box" testing, because every detail of the item to test (the "box") is accessible by the tester;
2. *Functional Testing*: it's a kind of "black box" testing, because the tester can access the item only to test through its external interface, and does not have access to its internal structure and behavior.

In the following sections, we describe in deeper detail Unit Testing and Functional Testing.

2.2 Unit Testing

Unit Tests are pieces of code that are created by developers to prove that another, related piece of code, does what it should do. For each software module the developer writes, she

also writes tests checking whether it produce correct results, in an efficient way. A unit is the smallest possible testable software component.

Unit Tests should be developed:

- For each unit: in a procedural language, a typical unit is a function. Each function that is coded is provided of one or more unit tests that verify if it works properly under every condition, or at least in the most likely conditions, when the possibilities are very many.
- Whenever a bug occurs: in this case, a test that produces the bug is written, then the bug is corrected. The test makes sure that bug is actually eliminated and that, in the future, it will not reoccur.

The aim of unit tests is to check that the code does what it is intended, so we need to test small, isolated pieces of functionality. In our case, we verify that the functions generate the correct answers from our test data.

2.2.1 Unit Testing concepts

To get the most advantage out of using tests, they have to be executed automatically. Automated tests can be executed quickly whenever the developer feels they are necessary. When writing Unit Tests, we have to consider four key concepts:

- **Fixture** is a set of variables, initialized to proper values, used as repeatable input data for the tests. Each time a test is run, its Fixture is re-initialized, because previous tests might have corrupted the data, 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 system will have many Fixtures, each of which will be used by many tests, in many ways.
- **Test Case** is a single test, or an atomic set of tests, defined on a Fixture and implemented in code, in such a way to be repeatable and understandable by other developers. Using a procedural language, a Test Case is implemented as a set of (test) functions, operating on common variables holding the Fixture.
- **Check** A Test Case stimulates a Fixture and checks for expected results. If a test is unsuccessful, it has to provide helpful information about the kind of error and about its location. 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. Examples of Checks are:

assert(boolean expression): the expression is evaluated, and the check is passed if it is true;

equals(expression1, expression2): the two expressions are evaluated, and the check is passed if their values are exactly the same;

equalsDouble(expression1, expression2): the two expressions are evaluated, and the check is passed if their values are equal within the precision of a double precision floating point value;

Checks help to organize checking for expected results. Usually, they can be associated to a message explaining what happened if the Check fails. Failing to pass one of such Checks is called a failure. In the case of unanticipated errors, they are dealt with depending on the specific testing framework and language used.

- **Test Suite:** Test Cases are grouped and organized in Test Suites, sets of Test Cases that can be run automatically in a repeatable way. A Test Suite, beside single Test Cases, can contain other Test Suites, providing a flexible grouping mechanism. A Test Suite should be easily stored, then brought in and run, allowing to compare results with previous runs. When a Test Suite is run, all its Test Case are run one after the other, and all failures and errors are recorded in a log file, or are shown to the developer in a window if the testing framework is provided of a graphic user interface.

2.2.2 Unit Testing in FLAME

In this section, we give some guidelines for designing and writing unit tests for FLAME. In EURACE system, the software units to test are mainly the X-machines themselves, considered as state machines, and the functions, written in C code, implementing the system behavior. Testing a state machine means to verify that state transitions are performed correctly from a given state to another.

We know that in FLAME, each X-Machine agent is defined by specifying:

1. **Memory:** the variables holding the data structure associated to the X-Machine.
2. **States:** the states of the X-Machine and the names and target states of their transition functions.
3. **Functions:** the transition functions associated to the states.
4. **Messages:** the messages that can be sent and received by X-Machines.

In order to test the behavior of each agent, we need testing its transition functions. A transition function is a function that makes an agent to act; it modifies the agent's state, and sends or receives message to/from others agents.

So, we can write one or more test cases for each transition function, starting from a predictable and repeatable configuration, or Fixture. For a transition function, a Fixture is an initial state of the agent under test, and of all other software entities used in the test. This means that it is necessary to set the initial states of the involved agents, and one or more input message in the message board. In this way, calling a transition function from a particular fixture should lead to a predictable state.

If the agent reaches the predicted state, the transition function is correct. A test case fails if the transition function activation on the Fixture creates a *failure* (unexpected result) and/or an *error* (syntax or run-time error). After we set the fixtures, we have to write the assertion functions on the output state and on the output messages of the agent.

Since FLAME is based on C language, the testing framework of choice is CUnit [4], a testing framework developed for C code modules. It offers a set of headers and libraries that helps to automatically execute user-defined tests, making easier to keep under control the system evolution.

CUnit provides a rich set of assertions. Each assertion tests a single logical condition, and fails if the condition evaluates to FALSE. The most common assertions defined by CUnit are:

- `CU_assert_TRUE(value)` Assert that *value* is TRUE (non-zero).
- `CU_assert_FALSE(value)` Assert that *value* is FALSE (zero).
- `CU_assert(int expression)` Assert that the *expression* is TRUE (non-zero).

For each X-Machine, the general guidelines to be used for designing and writing tests are:

- define one or more Fixtures representing the X-Machines to be used in the test. In other words, the Fixture is used to initialize a specific situation we want to analyze; for instance the agent memory or initial state and input messages;
- define one or more test cases for each transition function, starting from a repeatable configuration. Calling a transition function from a particular fixture will lead the agent to a predictable state;
- write the assertion functions on the output state and output messages of the agent, and check whether the function works properly;
- define test suites to group the test cases defined for each agent;
- run automatically all test suites for all agents.

2.3 Functional Testing

Functional tests are pieces of code that are typically created by testers, who are developers specialized in writing and executing tests. Unlike the case of unit tests, it is important that functional tests are not written by the same developer who wrote the code that is tested. Functional tests prove that the functionalities specified in the requirement specification work.

Functional tests are a kind of "black box" testing, because the tester can access the item to test only through its external interface, and does not have access to its internal structure and behavior. For this reason, functional tests are less dependent on the technology used to develop the software system than unit tests.

When we talk of functional testing, however, we should distinguish between tests performed on the system as a whole, as those referred to in the previous paragraph, and tests performed on a high level software module.

In the latter case, we want to test a complex module accessible only through its interface, but still part of the system under development. This is usually accomplished using the same testing framework of unit testing, for instance writing test functions in the case of a procedural language.

2.4 Automatic Testing Tools for EURACE

A good testing process can be an effective and practical way to facilitate the development of a large system subject to continuous changes and updates. As highlighted before, a suite of automatic tests developed together with the system can give a reasonable trust that modifications made to the system have made no harm to it, if the tests are all successfully run after each update. On the contrary, if some test fails, it is much easier to find the bug introduced and to fix it. In the absence of such an automatic test suite, a large system like EURACE

software framework will soon go out of control, and it will be made almost impossible for anyone to control system development.

To successfully control EURACE system development, we devised and applied a testing strategy to the software development process. Program testing involves each developer to write testing code. In order to support the writing of testing code, we developed two automatic tools to ease this task, which we present in this document.

The first tool is an unit testing tool specifically addressed to FLAME agent-based software developed in C language. It allows the writing of unit tests in a declarative way, able to be run automatically and in a repeatable way. In this framework, test cases are automatically generated and can run continuously without any external intervention. It is possible to associate one or more suites (sets of test cases) for each module of the EURACE system, according to the modular organization of the whole system. Each test suite is produced and executed automatically, and is described by a specific "markup" language.

The second tool is an automated testing tool for performing functional tests. It allows to check if the main consistency rules of the EURACE model are verified. The use of both frameworks is a good strategy both for the verification and validation of the EURACE model.

This automated testing framework is presented in detail in the following chapters.

3 Automatic Unit-Test Generator Tool (AGT)

The agent-based representation of the EURACE model is characterized by a several functions, that we need to definitely trust to work properly.

The smallest function for an agent could be considered its transition function. This function allows an agent to act, to change its state, and to communicate with other agents. So, the correct operation of all transition functions is a necessary condition for the correct working of the whole system.

The best way to test small parts of the system, such as functions and modules, is writing the unit tests with an automatic tool that collects them in one or more suites, and run them automatically. For a large software system it is a good practice to organize the system in different modules, which must be tested using an automatic tool in a way consistent with the designed modular organization.

The Automatic Unit-Test Generator Tool (AGT) is able to run a collection of suites, where each suite contains a collection of test cases. Each suite is related basically to a specific module, such as Financial Market or Credit Market module, etc. A module can be composed of many sub-modules, so we can build many suites, each one for a single sub-module, as a shown in Figure 2.

For instance, the Financial Market module is populated by agents such as Household, Firm, Clearing House, etc.; each one of these agents can be seen as a sub-module. For each of these sub-modules, we build a specific test suite, whose tests stimulate and check the behaviour of the related agent, or sub-module. In this way, we reach a good, modular organization of the system. Figure 3 shows the modular organization of the test suites for the Financial Market module. The set of the suites is collected in an *xml* file, that allows to configure the system and to group all suites.

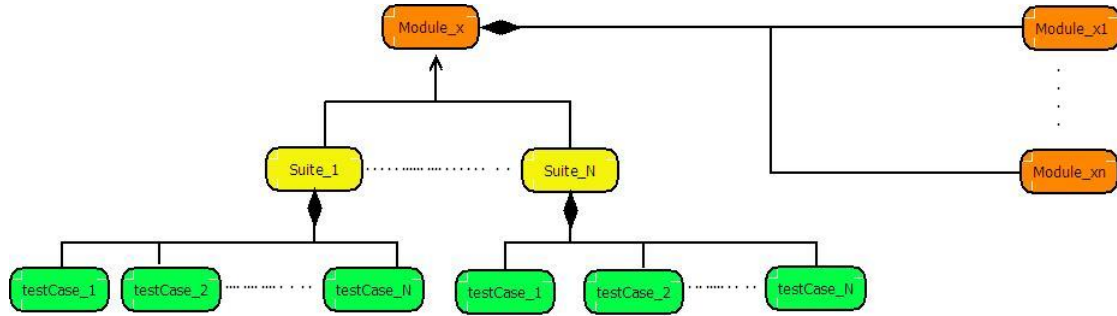


Figure 2: Modular organization of the test suites. There is a test suite for each sub-module of the system.

The main idea behind AGT tool is to hide from users the C code, allowing her to define the tests in a specific, XML-based language, and taking care of all the tasks related to C programming. Clearly, this does not mean that the user can ignore completely the C language. This language is in fact necessary to code the agent behaviour in functions, and to interpret and fix the bugs. However, being able to define tests without resorting to C code can greatly improve the productivity of writing tests. Moreover, the xml code defining the tests, if written following the convention to use self-documenting names, can improve substantially also the documentation and reuse of tests.

In the next sections, we describe the concepts and the key elements that characterize the

testing tool: *configuration file*, *suite*, *constants*, *unittest*, *declarative part*, and *fixture*. Appendix A reports a short guide on how to install and use AGT tool.

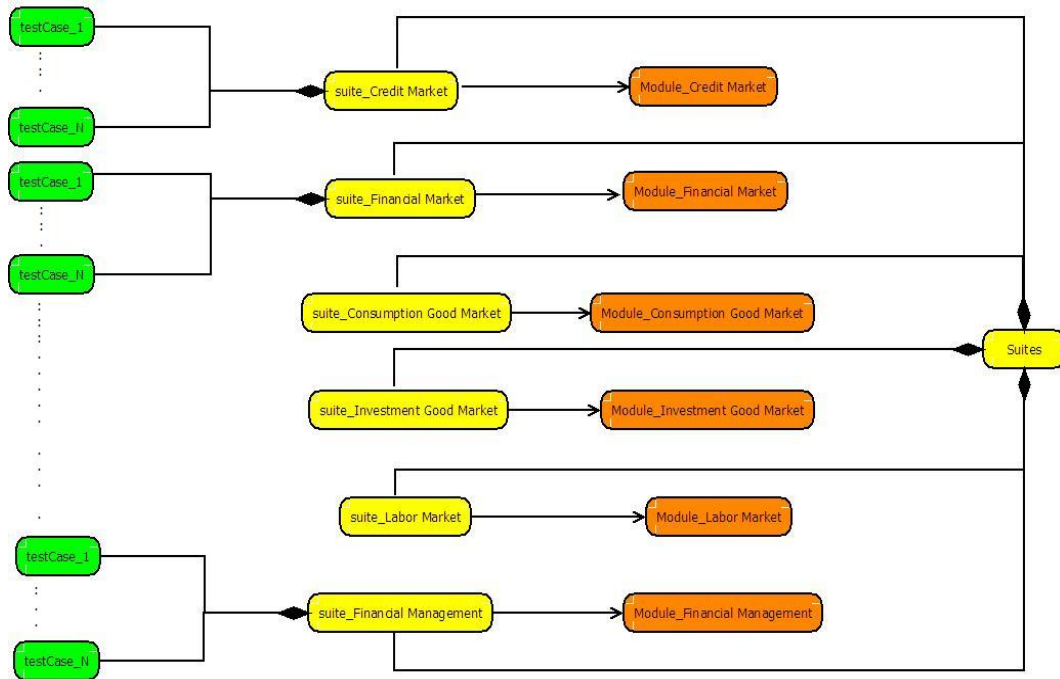


Figure 3: Modular organization of the test suites in the case of the Financial Market module.

3.1 The configuration file

The set of test suites developed for the whole model is declared in an *.xml* configuration file, which is used to configure the paths of all suites and the path of the model. This file is placed in the top folder of the model, the same folder that contains the related model description.

The example code shown in Figure 4 describes the configuration of a set of possible EURACE test suites, in this case related to its economic subsystems.

```
<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 4: An example of configuration file. of the suites

The configuration file contains two fundamental tags:

- the **modelPath** tag, that associates the testing process to the FLAME agent model;
- the **suitePath** tag, that allows to add a new suite related to a specific path.

3.2 The Suite

The suite is described by a file with extension *.xml*, whose name is declared in the previous configuration file, and which must be placed in the in the same folder of the tested sub-module. The suite is identified by a name and is characterized by a collection of unit tests.

The agent model is endowed with a list of constants that have to be specified. The declaration of a suite is opened with a tag called "*suite*". It contains the following tags:

- **name** tag: specifies the suite name. The name must be unique in the system.
- **constants** tag: specifies the values of each constants. We must specify only the constants that are used in the test cases collection. We will explain this in more detail in section 3.2.1.
- **unittest** tag: defines the unit test specification. This tag is repeated each time that we define a test-case.

3.2.1 Constants

The *constants* tag allows to set, for testing purposes, the values of the constants which are defined in the related agent model.

Moreover, we need also to set some constants specific for testing; these constants, for instance, specify the *seed* or the *current_day* variables.

The *seed* is a particular constant needed to set the seed of the random number generator implemented in the EURACE system.

The *current_day* represents the present day of the calendar. This constant is as important as the *seed* to get repeatable tests, because the behaviour of several functions changes accordingly to the calendar day.

Figure 5 shows an example of constants definition. In this example, we define two constants, (*firm_planning_horizon* and *seed*), giving a specific value to each of them.

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

Figure 5: An example of constants definition.

3.3 The Unit Tests

A unit test allows to define a test case. It is specified using the *unittest* tag. Several test cases can be written for a specific function, with the possibility of writing one or more fixtures for them, to cover as many as possible test cases.

The complete verification of a program, or of a part of a program, in any phase of the software life cycle can be obtained by performing the testing process for every element of the domain. If all test instances succeed, the program is verified; otherwise, one or more errors are found. This testing method is known as exhaustive testing and is a technique that guarantee the validity of a program. Unfortunately, this technique is not practical for real

systems. Often, functional domains are infinite, or even when finite, sufficiently large to make infeasible the number of required test instances. In order to reduce this potentially infinite exhaustive testing process to a feasible testing process, we must find criteria for choosing representative elements from the functional domain. The subset of elements chosen for use in a testing process is called a *test data set* (*test set* for short). Thus, the key part of the testing problem is to find an adequate test set, large enough to span the domain and yet small enough to allow the testing process to be performed for each element in the set. In AGT tool, the unit test is set by declaring the following items:

- the **name of the test**. It has to be unique in the system.
- the **name of the transition function** that is 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 are built.
- the **fixture**.
- the **expected values**.

3.3.1 The declarative part

The declarative part specifies the variables and messages that are subject to assertions in the test. The declarative part is opened using the tag called *declaration* and contains a list of variables and a list of messages.

The code shown in Figure 6 is an example of a typical declarative part.

The *xml* code shown in Figure 6 shows two variables and a message that are subject to assertions in the unit test generated. The **test_generator** will generate assertions related to the *assetsowned* and to the *pending_orders* variables, and to the *order* message.

3.3.2 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 usually have several fixtures, each of which will be used by many tests, in many ways. This part is opened with the tag called **fixture**. It contains a list of initialization values of agent's variables, and a list of the initialization values of messages. Figure 7 reports the code defining a typical fixture. Note that the *xml* tags of the fixture are the names of the variables defined in the previous section, and their values can be interpreted and assigned due to the type declared in the previous section.

```

<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>

```

Figure 6: Declarative part of the tool input.

```

<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>

```

Figure 7: The definition of a Fixture.

3.3.3 The expected values

A Test Case works on 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 its location. To this purpose, the system that launches the tests shows a summary of all tests, ordered by their 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 an expected output message. The *expected values* part of the test definition is opened with a tag called **expected_states**, and contains a list of the expected values of the agent's variables and a list of the expected values of output messages.

Figure 8 shows an example of the expected values of a typical fixture. It shows two variables holding the expected state of the Household agent. The `expected_state` part does not contain a list of expected messages, because the related transition function does not involve any output message.

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

Figure 8: An example of expected values of a test.

4 Automatic Integrity Testing Tool (AIT)

An important part of the validation process is to ensure that the structure of the EURACE model and its assumptions meet the purpose for which they are intended.

A way to obtain this is to address the automatic verification of high-level requirements of the EURACE model, defining a strategy to cope with conceptual validation. To this purpose, we built a tool able to check automatically the integrity of a set of internal consistency rules of the model.

For instance, it is possible to define some accounting rules to assess the consistency of the model monetary flows. Other rules can also check the consistency of portfolio sizes versus the total number of stocks exchanged in the model since the beginning of the simulation. More information can be found on this subject in the EURACE report [6] a stock-flow consistency model was defined. Verification of consistency rules is an important step of the system validation phase because it supports the model accreditation process and helps to raise trust on the model's results.

The automatic testing tool we developed, called **Integrity Testing Tool (ITT)**, allows to test the consistency rules of EURACE model and to verify if the integrity of some resources are preserved. If the consistency rules are not verified, we could go back to the possible causes, for instance bad transactions or bad computations, making use of:

- the visualization GUI;
- debugging techniques;
- automatic unit test suites.

The combination of all these approaches will help us to deliver a reliable and usable system.

The rationale of AIT tool is similar to that of AGT – to provide the user a way to code tests efficiently using a xml-based language, avoiding direct coding in C language or, in the case of consistency checks, the use of a post-processor on generated simulation data. Note that the data generated by a simulation can be of many GByte, so that visual inspection is out of question. Note that AIT works exclusively on the output files of the simulations. It does not perform consistency checks during the simulations, but is able to analyse the xml files produced by EURACE simulator, mainly verifying that the values of declared parameters, and of sums of parameters satisfy certain constraints against other values, or sums of values.

In the following subsections we describe in deeper detail AIT tool, while in Appendix B we report a short guide on how to install and use this tool.

4.1 Consistency rules of EURACE model

EURACE simulations produce an output file for each simulated *iteration* declared as output. This means that one can have an output every n iterations. The output file holds all the information about what happened in the related iteration, for each X-Agent involved (payments made and received, stocks bought and sold, and so on). The parameters named in the output file, that correspond to data of an Agent, or to other activities performed, can be used in the rules. A consistency rule always refers to the sum of all the named parameter values, computed on all the agents of one or more named type, and not to the value of single parameter.

For instance, let us consider three kinds of X-Agents, Bank, Firm and Household, and let us suppose that Households and Firms deposit their own payment accounts into the Bank. In an economic system, the following rule about Bank deposits must be verified:

$$\sum_{\substack{1 \leq i \leq M \\ 1 \leq j \leq N}} \text{payment_account}(\text{Household}_i, \text{Firm}_j) = \sum_{1 \leq k \leq P} \text{deposits}(\text{Bank}_k) \quad (1)$$

This means that the sum of all payments made by both Households and Firms to their bank accounts must be equal to the sum of the deposits received by all Banks, in any given iteration. Here indexes i , j and k refer to Household, Firm and Bank, respectively, being the total number of Households, Firms and Banks M , N and P , respectively.

Another consistency rule regards Bank and Central_Bank X-Agents, supposing that the Central_Bank loaned money to the Banks. The rule could be expressed as following way:

$$\sum \text{ecb_debt}(\text{Bank}) = \text{total_ecb_debt}(\text{Central_Bank}) \quad (2)$$

Here we do not explicitly report the sum indexes. The right hand-side of equation 2 does not report a sum because there is only a Central Bank in the system.

A third example rule can be obtained by considering Banks and Firms, checking if the total credit of the Banks is equal to the total debt of the Firms:

$$\sum \text{total_credit}(\text{Bank}) = \sum \text{total_debt}(\text{Firm}) \quad (3)$$

Generally speaking, all rules constrain the sums of one parameter referring to one or more kinds of agents. A rule is always about relating two sums, but can also relate a sum of sums. For instance, the sum of all wages of Households must be equal to the total labour cost of Firms and IG_Firms (which are different kinds of agent):

$$\sum \text{wage}(\text{Household}) = \sum \text{labour_costs}(\text{Firm}) + \sum \text{labour_costs}(\text{IGFirm}) \quad (4)$$

A list of consistency rules is shown in Table 1.13 of the document [6].

4.2 The grammar rules of AIT

To implement in a synthetic and efficient way the integrity rules of **AIT** tool, we defined a basic grammar. This grammar specifies how to write the various lexical tokens expressing the

rules. In deeper detail, the grammar definition is:

```

<letter>::=_ | [a..z] | [A..Z]
<digit>::=1|2|3|4|5|6|7|8|9|0
<identifier>::=<letter>|<identifier><digit>|<identifier><letter>
<tag>::=<identifier>
<specifier>::=<identifier>
<specifierlist>::=(<specifier>,<specifierlist>)|(<specifier>)
<rule>::=<expression> = <expression>;\\
<expression>::=<expression> + <expression> | <tag>| <tag><specifierlist>

```

Some example statements that satisfy the grammar are shown in Figure 9:

```

nr_gov_bonds = nr_bonds_outstanding;
nr_gov_bonds(Household,Firm) = nr_bonds_outstanding(Government);
wage(Household) = labour_costs(Firm) + labour_costs(IGFirm)
current_shares_outstanding (Firm) = constant

```

Figure 9: Some integrity rules written accordingly to lexical-grammar of AIT. rules

4.3 Specifying integrity rules

Actual integrity rules are specified in xml file "rules.xml", located in the integrity source directory.

This file is composed of three parts:

- Declaration of the directory where to find output files to analyze.
- Declaration of the iterations to be considered for checking.
- Integrity rules.

Figure 10 shows a simple example of "rules.xml" file.

After setting the path of the folder where simulation output files reside, the iteration numbers to consider are given. Eventually, some integrity rules are declared.

Rule:

```

monthly_consumption_expenditure(Government,Household)
    = cum_revenue(Firm);

```

computes, for each iteration, the sum of *monthly_consumption_expenditure* parameter, computed over all Government and Household agents of the simulation, the sum of *cum_revenue* parameter, computed over all Firms, and then checks the equality of these sums.

Rule:

```

wage(Household) = labour_costs(Firm) +labour_costs(IGFirm);

```



```

<integrity>
  <directory>tests/</directory>
  <iteration>
    <start>0</start>
    <stop>100</stop>
  </iteration>
  <rules>
    <rule>monthly_consumption_expenditure(Government,Household)
      = cum_revenue(Firm);</rule>
    <rule>capital_costs(Firm) + monthly_investment_expenditure(Government)
      = cum_revenue(IGFirm);</rule>
    <rule>wage(Household) = labour_costs(Firm) +labour_costs(IGFirm);</rule>
    <rule>payment_account(Household,Firm,IGFirm) = deposits(Bank);</rule>
    <rule>payment_account(Household)+deposits(Bank)=dummy;</rule>
  </rules>
</integrity>

```

Figure 10: An example of integrity rule declaration.

computes, for each iteration, the sum of *wage* parameter, computed over all Household agents of the simulation, the sum of *labour_costs* parameter, computed over all Firms, and the sum of *labour_costs* parameter, computed over all IG_Firms. Then it sums the latter two sums, and checks the given equality.

Rule:

```
payment_account(Household,Firm,IGFirm) = deposits(Bank);
```

computes, for each iteration, the sum of *payment_account* parameter, computed over all Household, Firm and IG_Firm agents of the simulation, and the sum of *deposits* parameter, computed over all Banks. Then it checks if the two sums are the same.

Rule:

```
payment_account(Household)+deposits(Bank)=dummy;
```

computes in its right-hand side, the sum of *payment_account* parameter, computed over all Households, and the sum of *deposits* parameter, computed over all Banks. Then it cumulates the two sums, and checks them against the *dummy* keyword. In fact, when AIT tool parser encounters an unknown word, such as *dummy* in this case, it considers it as zero, and usually the equality is not verified. This rule is a shorthand to force the computation of the sums, and to show them in the output file.

4.4 Application output

Figure 11 shows the results of the application of integrity rules reported in Fig.10 for iteration 1 of the simulation.

The first rule is verified, being the sum of monthly consumption expenditures of Government and Households equal to the total revenue of Firms. The actual values of the sums are reported in the output, as well as the difference between the sums, which is zero.

```

ITERATION 1
+ monthly_consumption_expenditure (Government,Household) =1.214560
- cum_revenue (Firm) =1.214560
  THE RULE IS VERIFIED 0.000000

+ capital_costs (Firm) =2.862754
+ monthly_investment_expenditure (Government) =0.000000
- cum_revenue (IGFirm) =0.000000

  THE RULE FAILED 2.862754

+ wage (Household) =36.000000
- labour_costs (Firm) =36.000000
- labour_costs (IGFirm) =0.000000
  THE RULE IS VERIFIED 0.000000

+ payment_account (Household,Firm,IGFirm) =15859.253147
- deposits (Bank) =15859.252575
  THE RULE FAILED 0.000572

+ payment_account(Household) =7984.368349
+ deposits(Bank) =15859.252575
- dummy =0.000000
  THE RULE FAILED 23843.620924

```

Figure 11: An example results due to integrity rule application.

The second rule failed, because the application found that the Government monthly investment expenditure is zero, while the capital costs of the Firms is greater than zero. This may denote an error in the model or, as in this case, an error in the integrity rule.

The third rule, regarding wages and labour costs, is verified. However, the output highlights that in this simulation and in this iteration the labour cost of IG_Firms is zero. This can denote a problem, or simply the fact that in this simulation there was no active IG_Firm.

The fourth rule fails, but its failure is clearly due to a rounding problem. This might denote an error in the model, or in the model algorithms, or be of no importance.

Finally, the last rule fails because it is not a "true" rule, but a shortcut to get the values of the sum of the *payment_account* values of Households, and of *deposits* values of Banks.

A Using Automatic Unit-Test Generator Tool

In this Appendix we describe the steps to set up, to generate the C code of unit test, and for running the test suites automatically.

A.1 Set up of the tool

To install AGT, you need first to properly install the *libxml2* and *CUnit* libraries, that can be downloaded from "http://xmlsoft.org" and "http://cunit.sourceforge.net" sites, respectively.

AGT can then be downloaded from the EURACE repository, at the address: "http://ccpforge.cse.rl.ac.uk/svn/EURACE/tests/unit-test". Here you can get the unit-test-generator folder, containing the following files: "Makefile", "main_code.c", "datatype.c.tmp", "header.h.tmp", "Suite.h.tmp", "Suite.c.tmp", "messages.c.tmp", "LauncherTest.c.tmp". The Makefile will take care of AGT compilation and installation.

The test_generator is used to generate the code of unit-test, using the information contained in the EURACE model, and the suite declaration. The test_generator builds a platform to launch the agent transition functions, which are defined for each agent. This is obtained by creating a specific environment and its "objects", with the same interface built into the FLAME project. The AGT environment includes a pseudo-messageboard, a pseudo-list of agents, and a template for generating datatypes and dynamic arrays.

The generated code is composed of some C code files, and of a Makefile. This set of files is the testing system, which we can launch by executing the "LauncherTest" program.

Figure A.1 schematically shows the inputs and the outputs of test_generator. The output files produced are placed in the same folder of the agent model. These files have to be then properly compiled.

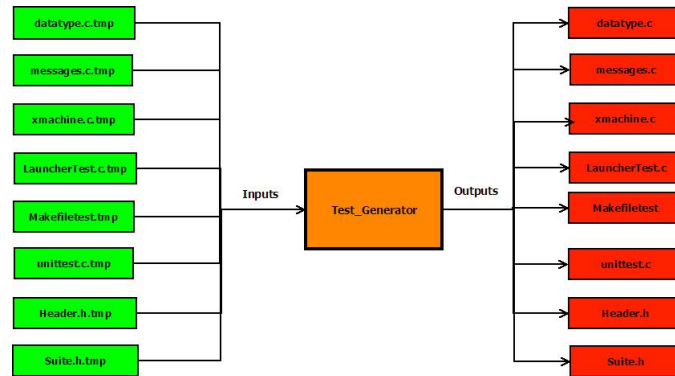


Figure 12: Inputs and outputs of the test_generator

Before you can use test_generator, you have to run the make command on the **test_generator**:

1: \$ *make*

This may take a while, depending on the load of your system and on your file server. It may take anywhere from a few seconds to a minute or more.

A.2 Test running

When the AGT framework is properly installed, you have to write the test definition, to be put in a *suite.xml* file, accordingly to the guidelines given in Section 3:

- 1: \$ `cd ../unit-test/unit-test_generator/`
- 2: \$ `./test_generator path_of_model/suites.xml`

The directory path, identified by *path_of_model*, contains the agent model (typically it is a *model.xml* file), and the test suite configuration file.

At this point, you have to create the system of test for your agent model, which has then to be compiled. This procedure will produce a set of C files and make files.

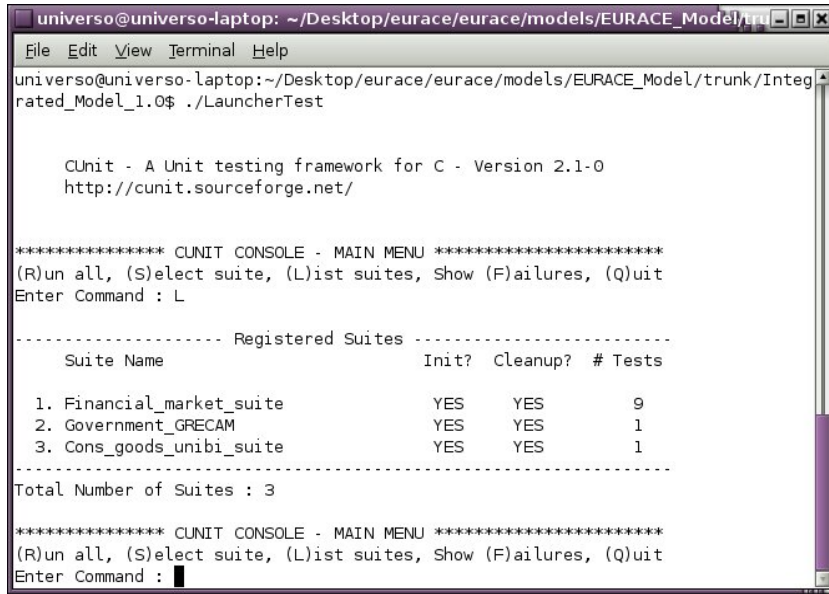
In practice, you must run the *make* command on the test system:

- 1: \$ `cd...path of model`
- 2: \$ `make -f Makefiletest`

Finally, after you have run the *make* command on the system, you have to run it using the following command:

- 1: \$ `./LauncherTest`

Running the command above, a CUnit test console will appear, as shown in Figure A.2. From this console, you can launch all tests, or a single suite.



```
universo@universo-laptop: ~/Desktop/eurace/eurace/models/EURACE_Model/trunk/Integrated_Model_1.0$ ./LauncherTest

CUnit - A Unit testing framework for C - Version 2.1-0
http://cunit.sourceforge.net/

***** CUNIT CONSOLE - MAIN MENU *****
(R)un all, (S)elect suite, (L)ist suites, Show (F)ailures, (Q)uit
Enter Command : L

----- Registered Suites -----
Suite Name                               Init?  Cleanup?  # Tests
-----
1. Financial_market_suite                 YES    YES       9
2. Government_GRECAM                     YES    YES       1
3. Cons_goods_unibi_suite                 YES    YES       1
-----
Total Number of Suites : 3

***** CUNIT CONSOLE - MAIN MENU *****
(R)un all, (S)elect suite, (L)ist suites, Show (F)ailures, (Q)uit
Enter Command : 
```

Figure 13: Unit test console

B Using Automatic Integrity Testing Tool

In this Appendix we describe the steps to set up and to install the Automatic Integrity Tool, in order to facilitate its use.

B.1 Set up of the tool

The first step for setting up AIT tool is downloading it, and installing all the needed files for using it.

You can download AIT from the EURACE repository, at the address: "<http://ccpforge.cse.rl.ac.uk/svn/EURACE/tests>". From this folder, copy the entire "integrity_tests" sub-folder to a proper folder of your computer.

Before you can use AIT, you should run the make command on the **Integrity** sub-folder:

```
1: $ cd Integrity
2: $ make.exe
```

In Linux, the command is *make* and not *make.exe*. This command may take a while, depending on the load on your system.

B.2 Running integrity tests

Before running an integrity test, you need to write some consistency rules in an .xml file called *rules.xml*. Thereafter, you must create a .txt file where to print test results. For instance, you can create a file called *results.txt*.

The next step is to set the path of the folder where you have got all simulation outputs in the *rules.xml* file. Eventually, you can open a shell and run the AIT on your system. To run the AIT use the commands:

```
$ cd ../integrity_tests $ ../tests/rules.xml results.txt
```

In this way, you have run the declared integrity test on your system. All test results are printed in file called *results.txt*.

References

- [1] David Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall, 2003.
- [2] Kent Beck. *Manifesto for agile software development*. 2001.
- [3] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
- [4] Paul Hamill. *Unit Test Frameworks*. O’ Reilly Media, 2004.
- [5] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Signature Series, 2007.
- [6] Sander van der Hoog, Saul Desiderio, Marco Raberto, Andrea Teglio, and Philipp Harting. *System of National Accounts in EURACE*. 2009.