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 modeling and simulation

Instrument STREP

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

Deliverable reference number and title

D1.2 Agile methodologies for defining and testing agent-based models.

Due date of deliverable: August 31st 2007
Actual submission date: September 26th 2007
Organization name of lead contractor for this deliverable
University of Cagliari - UNICA

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)	

Contents

List of Figures

1 Executive Summary

This technical reports describes some of the software engineering approaches and practices which have been studied with the aim of improving the quality and efficiency of the implementation of large economic systems using the X-Agent FLAME framework, in the EURACE project.

It is logically divided in three parts, describing:

1. the software development process;
2. the analysis and design of X-Agent systems;
3. the automated testing practices.

The software development process proposed for EURACE project is an agile process, the best known approach to develop systems with changing requirements. In fact, being EURACE a research project, we expect that new models and new ideas are developed throughout the project. Consequently, the software simulator implementing these models needs to be able to change continuously, keeping a high quality.

Agile Methodologies (AMs) for software development were introduced in the end of the nineties to address the problem of changing requirements and software quality, and are now used by many software projects. The most popular among AMs are Extreme Programming (XP), Feature-Driven Development (FDD), and Scrum.

In the EURACE context, we propose a very lightweight, distributed AM based on practices derived from Scrum and XP, called EURAM (EURace Agile Methodology). EURAM addresses the problem of parallel distribuited software development in research units that are located in different countries and communicate throught the Internet. The key principles behind EURAM are: (i) being very simple; (ii) being able to adapt to changing requirements; (iii) being able to exploit distributed development; (iv) being able to deliver working software at a constant pace, throughout the duration of the project. We define developers' roles, activities, artefacts and meeting of EURAM process.

The second part of the report deals with analysis and design of X-Agents, which are software concepts bearing some similarities, but also many differences, with software objects. We present the detailed steps needed to design X-Agent systems, and propose to use a notation derived from UML (Unified Modeling Language) to graphically describe an X-Agent system. In particular, we define the Agent-Message Diagram, derived from UML Class Diagram, able to give a graphical representation of X-Agents, messages and their relationships.

The third part of the report presents how to take advantage of automated testing techniques even in the context of X-Agent programming using the

FLAME framework. Using the popular testing frameworks named *C-Unit*, we propose a testing methodology and give detailed examples on how to design and write tests cases for ADT (Abstract Data Types) defined in C and for X-Agents developed with the FLAME framework.

2 Software Development in EURACE

The main goal of EURACE project is to develop a software simulator of European economy. Beside the huge amount of work, to be performed by economists, to devise sound models of the many interacting components of a modern economy, and of their interactions, the other main part of EURACE is a large software development project. As every large software development project, it must be properly managed, using practices and tools to manage the various phases of the development (requirements elicitation, analysis, design, coding, testing, deployment), and the overall development process itself. EURACE software environment is not conventional, because it makes use of software agents, and is aimed to be run on massively parallel computers. Moreover, the development is made by various project units which are not co-located, but which need to be coordinated and work together using the Internet. For these reasons, EURACE software development needs a specific process management methodology that has to be specified and followed by the various teams actually performing the development. This section of the report deals with this software development process. In modern software engineering, there are two main approaches to software development, the so called "planned" and "agile" ones. The planned approach is more traditional, and can be successfully applied when the system requirements are well defined and stable. If this is the case, it is sensible to spend a substantial amount of resources to build up-front a detailed analysis and design model of the system to be built. In this way, the requirements are gathered, verified and specified only once, and then the system coding can proceed smoothly. Unfortunately, clear and stable requirements since the beginning of the project are seldom present in the realty of software development, and certainly this is not the case of EURACE project, whose requirements will depend on the specific economic models developed throughout the project, and on other ongoing activities of user interface specification and interaction with European Bodies. On the other hand, starting software development only when the requirements are understood and stable would mean to start in the last months of EURACE, which would clearly result in not being able to successfully complete the economy simulator. For these reasons, the best approach to software development in EURACE is the use of the so called Agile Methodologies (AM), a recent approach to the software process able to manage unclear and changing requirements. Agile processes are a set of processes, practices and tools for software development that have emerged since 1990. They are centered on close involvement and frequent communications between the developer team and business experts, and on steady delivery of functionalities. In particular, Agile processes use iterative, incremental techniques and relies on self organizing, self managing teams. The "Agile

Manifesto"[1] establishes a common framework for these processes, because it values: individuals and interactions over processes and tools;

- individuals and interactions over processes and tools;
- working software over comprehensive documentation;
- customer collaboration over contract negotiation; responding to change over following a plan [Bec01].

In anticipation of changing requirements, agile processes calls for projects to be broken down into smaller, bite-sized modules or iterations, each of which is worked on as a separate entity. The customer is also tapped for knowledge along the way to ensure that any changes in requirements can be applied at the earliest stage. There are various development methods that fall under the agile banner: Extreme Programming (XP) [Bec00], Lean Development [PM03], Scrum [Coh03][Sut03], and many others. All these methodologies were devised for software development teams working together and collocated, and most of them were born with object-oriented development in mind. Since EURACE software development is not collocated neither using object-oriented programming, we decided to propose an AM specific EURACE, in the spirit of agility, which assumes to be proactive and able change the development process when needed.

The proposed AM, which we will name EURAM (EURace Agile Methodology) from now on, has been derived mainly from Scrum, XP and FDD:

- it takes the basic roles and artifacts from Scrum;
- it takes the iterations and process control mechanisms from XP (and to some extend also from Scrum);
- it takes the need of an upfront analysis and design, albeit synthetic, from FDD; this feature is needed because working with X-Agents, and with their FLAME implementation, based on C language, the agility reached with modern object-oriented languages and their interactive programming environments cannot be achieved; hence, an accurate design phase is needed before starting the coding activity.

The next chapter introduces EURAM, describing and motivating its roles, practices, meetings and artifacts.

[1]see:www.agilemanifesto.com

3 The EURAM software development process

The key principles behind EURAM are the following:

1. being the simplest process that can possibly work in a distributed, agent-based, C language development;
2. being able to adapt to changing in requirements and in underlying technology;
3. being able to exploit distributed development, trying to keep structured interactions among teams at a minimum;
4. being able to deliver working software at a constant pace, throughout the duration of the project.

As any other AM, EURAM is iterative and incremental as proceeds through iterations. It is based on roles, artifacts, meetings and practices. We will introduce all of these concepts in the following sections, leaving the overall view of the process at the end of the chapter. Note that, since in EURACE there are more than one software development projects, and namely those happening in Workpackages WP1 and WP8, each of these will be managed by an instance of EURAM methodology, provided that proper interchange mechanisms have been devised for the two projects.

3.1 EURAM Roles

The roles of EURAM are:

1. **Project Manager** (Product Owner in Scrum): this person has in control the whole software development project, and controls that the artefacts delivered by the process are in line with the research project aims and the required deliverables. A natural candidate for this role is the coordinator of the Workpackage responsible for the software development project.
2. **Process Manager** (Scrum Master in Scrum, Coach in XP): in a distributed research project, this role is a person who is responsible for enforcing the rules of the process, helping to remove impediments and ensuring that the process is used as intended. A natural candidate for this role is a person expert in agile methodologies, belonging to the unit which is in charge of the software engineering process.
3. **Unit Leader**: this is a new role, specific of a distributed research project. It is a person who coordinates the Team members belonging to a research unit in charge of software development and controls that the artefacts delivered by the unit are in line with those required. The

Unit Leaders, together with the Project Managers, decide the feature prioritisation.

4. **Team**: the software developers actually working on the project. Each team member belongs to a Research Unit.

5. **Stakeholder**: in general, this role refers to any person able to influence the requirements of the software project. In EURACE, natural candidates for this role are the Project Coordinator, the economists ruling the models to be implemented (mainly Research Unit coordinators), and possibly external stakeholders belonging to the Steering Committee and Advisory Board, and to the European Union

3.2 EURAM Artefacts

The main artefacts of EURAM are:

1. **Feature List** (Project Backlog in Scrum, prioritized User Stories in XP): a prioritised list of software requirements with estimated times to turn them into completed system functionality. Any Unit Leader can add features to the Feature List (or any Team member empowered to do so by his Unit Leader). Each feature must be implemented in a reasonable time (two or three iterations at most); if not, it must be splitted in shorter features. The Project Manager, together with the Unit Leaders, prioritizes the List. In the case they do not agree, the Project Manager decides.

2. **Iteration List** (Sprint Backlog in Scrum): a list of tasks that defines a team's work for an iteration of the project. The features to be implemented, or worked on, in the iteration are divided into tasks. Each Team owns a subset of tasks, and decides which developer will work on which task. Only the Team owning a task is allowed to modfy the task.

3. **Impediment List**: a list of anything around the software development project that impedes its productivity and quality. It includes blocks and unmade decisions. It is owned by the Process Manager and is updated often (at least once in a week).

4. **Increment**: at the end of every Iteration the Team should have delivered a production quality increment of the software system, demonstrated to the Project Manager and, at each project review, to external reviewers.

5. Burndown Graphs: they show the trend of work remaining across time in an Iteration, a release or the whole project. Only completed, working

features can be added to the Burndown Graph, and not completed tasks belonging to features still to complete.

6. Other reports on the project status, the bug status, etc., as needed.

Owing to the distributed nature of EURACE project, all these artefacts are electronic documents put on a Wiki, shared by all project research units.

3.3 EURAM Meetings

The main meetings of EURAM methodology are listed in the followings. Besides the intra-Team meetings, many of them have to be held online, due to the distributed nature of the development.

1. **Iteration Planning Meeting-IPM** (Sprint Planning Meeting in Scrum): a meeting held online before any iteration, where the Features List is reviewed and possibly revised, and the features to implement in the forthcoming iteration are decided and subscribed by the various Teams. A preliminary Iteration List is prepared, holding all the features to be implemented, or worked on, during the iteration. The IPM might be held interactively (for instance using Skype), but also not in real time through a Wiki. In the latter case, is important though that the IPM is time-bounded, and does not last more than one day.

2. **Iteration Team Meeting-ITM**: it is a meeting held within the Team to discuss the features the Team must implement during the iteration. Its output is the detailed list of task to be worked on by the Team, which is sent to the Project Manager to be merged with all other tasks coming from the other Teams, yielding the final Iteration List.

3. **Weekly meeting**: it is a meeting held every week online to discuss about the status and the impediments of the current iteration.

4. **Iteration Review Meeting**-IRM (Sprint Review Meeting in Scrum): a meeting held after each iteration, where the Increment produced during the iteration is discussed, and the new IPM is planned.

Other meetings (for instance a daily stand-up meeting, as in XP) can be planned and held within every Team, according to the Team's wishes. During the EURACE project meetings, the software development Teams will possibly hold a IRM and/or a IPM.

3.4 The EURAM Process

A short description of EURAM process is the following:

1. EURAM is based on specification of what has to be done through a list of features; each feature is assigned an estimated value for the project, an estimated effort, a responsible Team;

2. Project advancement occurs using time-boxed iterations. The duration of iterations is decided during the Iteration Planning Meeting. It should be between 2 and 4 weeks. There is no need that an iteration has the same length of the previous one (albeit this is desirable), but it is important that, once decided, the length does not change.

3. The Feature List is prioritized according to feature's value, effort and risk. This is done by the Project Manager, together with Unit Leaders.

4. Each iteration implements a subset of the Feature List with highest priority.

5. Iterations are grouped in releases. A release covers one year of work, and delivers project official deliverables.

6. At the beginning of each iteration, an Iteration Planning Meeting decides how long the iteration will last, what features to implement in the iteration and which Teams are responsible of them. The IPM is held through the Internet or, when possible through a physical meeting. The Process Manager shows the Feature List and invites discussion. As the Teams and Project Manager discuss priorities and details, the Process Manager updates the item directly online. Another solution that works much better when the meeting is co-located is to create index cards and put them up on the wall (or a large table).

7. Each Team meets during the Iteration Team Meeting, to discuss the features to implement during the iteration. these features are decomposed into tasks. Each task is assigned an estimated effort and which Team member is responsible for doing the work. The list of tasks is sent back to the Project Manager, to be merged with all other tasks into the Iteration List.

8. During the iteration, a Weekly Meeting is held every week, through the Internet. During this meeting the team members synchronise their work and progress and report any impediments. Other meetings among different teams are done via instant messaging, when needed. Other meetings within each Team can be held, as decided by Team's members.

9. An Iteration Review Meeting provides an inspection of project progress at the end of every iteration. based on the inspection, adaptations can be made to the project features, estimates and prioritization. At the end of the IRM also the process itself can be discussed, and possible

improvements can be introduced. The IRM is held through the Internet or, when possible, through a physical meeting. It typically precede the IPM deciding about the next iteration.

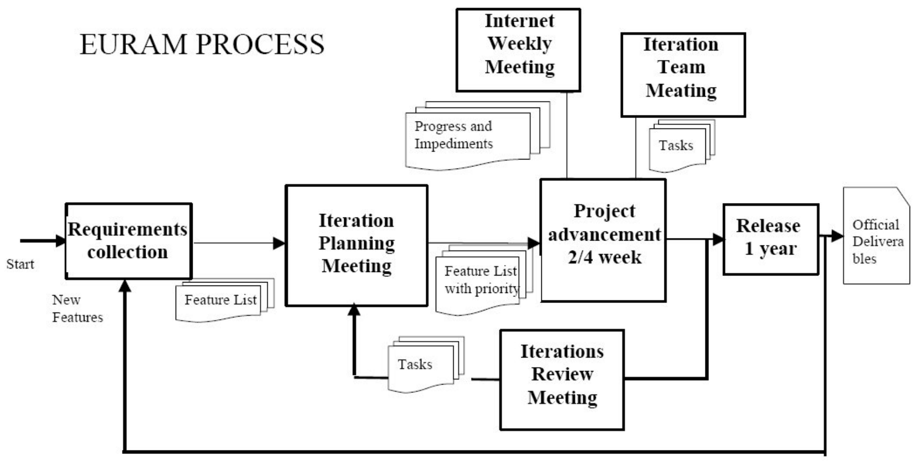


Figure 1: The EURAM process

4 X-Agent Analysis and Design

Microeconomical modelling of an entire economy entails very large and complex models, that are reflected in the underlying software system. A consequence of this fact is that the software system cannot be coded without a thorough analysis and design phase, able to manage its complexity, resolve possible inconsistencies and contradictions, and guide the development. Since, as said before, the exact requirements of the software system cannot be known in advance, one has to use an agile methodology, but this does not mean that analysis and design phases cannot be skipped. On the contrary, they must be performed following an analysis process able to be splitted into interactions. To this purpose, having a clear analysis and design process, and using a graphical notation that can be convey the information in a more immediate and readable way is of great importance. This is even more important using the computational paradigm of X-Machines, that lacks some of the complexity management mechanisms of object-oriented programming, and are thus more difficult to use when building complex systems. Here we will not describe yet another time what an X-Agent system is, and how it works. To this purpose, Deliverable D1.1 and the literature are more than enough. We only briefly outline the main characteristics of an X-Agent system. The features of a single X-Agent are:

- it has a memory, holding basic C types and structures, including an unique identifier among all agents of the system;
- it is a state machine, thus having admissible states and a starting state;
- it can execute a set of available functions (written in C);
- its functions can read and write the X-Agent's memory;
- its functions can read and write messages onto an external list (blackboard), publicy available to all X-Agents; this constitutes the input or output activity used to communicate with other agents;
- state transitions are managed by the functions that can (and will) be executed in each state, and by their behaviour, which depends also on internal memory and external messages;
- there can be precedence constraints on function execution.

The triggering of a state change is therefore not due to an external event, but to the execution of a function, provided that its inputs (X-Agent's memory and input messages) allow the transition. When an X-Agent is in a given state, one or more functions are always called, albeit these can also do nothing, if some preconditions are not met. The messages are the other key concept in X-Agent systems, and have the following characteristics:

- there may be many kinds of messages;
- there is a message list for each kind of messages;
- a message is basically a record, holding different pieces of information;
- a message may include the identifier of the agent it is directed to, and/or other information able to discriminate its receivers (for instance, the geographic coordinates of its "region", or the group number of agents it is directed to).

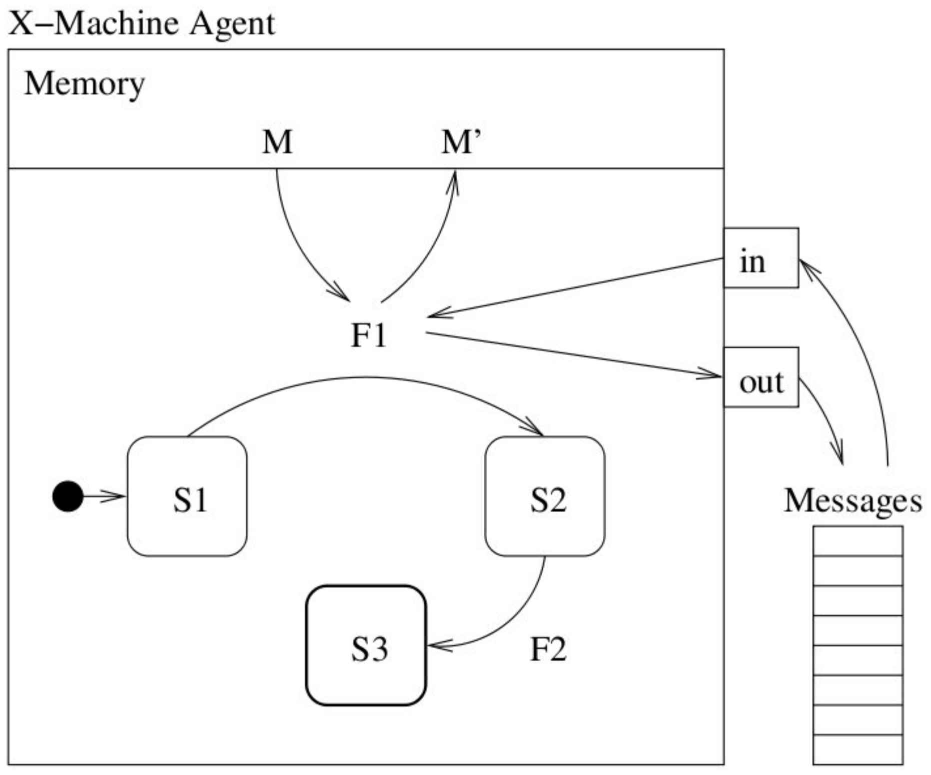


Figure 2: The structure of an X-Agent

The basic design process of X-Agents is already described in Deliverable D1.1, with many examples, and we will not deal it in detail. In short, after the mathematical and economic models to be implemented are decided, and key X-Agents are figured out, for each Agent one has to follow the following stages:

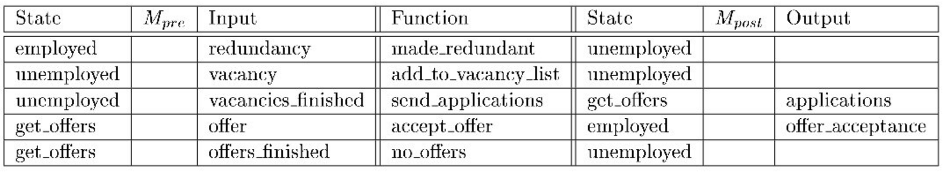
1. identify the agent's functions;

2. identify the states of the X-Agent, their transitions and those which impose some order of function execution;
3. identify input messages and output messages;
4. for each state identify the memory as the set of variables that are accessed by outgoing and incoming transition functions.

This activity leads to devising for each X-Agent a table whose rows are identified by a pair state-function, and hold also the preconditions for executing the function, the inputs consumed by the function, the end state after execution (that can be equal to the starting state), the effects on the agent's memory and the output generated by the function. The columns of this table are:

1. **Start state**: the state the row refers to.
2. **Memory preconditions**: a Boolean expression using X-Agent's memory fields and literal constants that must be true for the function to be executed.
3. **Input**: the message(s) read by the function.
4. **Function**: the name of the function called in the start state. More functions can be called in the same start state, leading to different rows of the table.
5. **End State**: the X-Agent's state after the execution of the function. It can be the start state itself.
6. **Memory after the computation**: the value of the relevant fields of the agent's memory resulting from the function execution.
7. **Output**: the message(s) written by the function in the proper message list(s).

Figure 3 shows an agent design table, taken from D1.1. In this simple example, the agent's memory is not used in the computation. More complex and comprehensive examples can be found in D1.1.



State	M_{pre}	Input	Function	State	M_{post}	Output
employed		redundancy	made_redundant	unemployed		
unemployed		vacancy	add_to_vacancy_list	unemployed		
unemployed		vacancies_finished	send_applications	get_offers		applications
get_offers		offer	accept_offer	employed		offer_acceptance
get_offers		offers_finished	no_offers	unemployed		

Figure 3: The table with states, functions and transitions of an X-Agent representing an Household (from Table 6, EURACE Deliverable D1.1)

5 A Graphical Notation for X-Agent Systems

In this chapter we present a graphical notation derived by UML and specifically aimed to represent X-Agent system. In software engineering, and in engineering design in general, graphic models are very used because they facilitate communication among developers, and between developers and stakeholders. Often, inspecting one ore more drawings is more effective than going through many tens of pages of a document. The Unified Modeling Language (UML) is the acknowledged standard for graphical documentation of software systems. Though it was initially devised for object-oriented systems, and works better with this kind of technology, its portfolio of different diagrams is widely used also for more conventional, structured programming languages and systems. We believe UML can be effectively used also for the documentation of X-Agent systems. Note that there already has bee a huge activity in proposing UML extensions to describe agent-based systems (see for instance [BB01] and [BB05]). However, most of this activity deals with conventional software agents, that are in general entities much more complex than an X-Machine, and are characterized by goals and intentionality, plans, beliefs, and exhibits complex interactions with other agents and the environment. Our effort, on the other hand, has the only goal to facilitate the analysis, design and implementation of X-Agents, that are very specific kinds of software agents. The UML, or UML-like, diagrams that are most useful for representing X-Agent systems are:

- State Diagrams
- Activity Diagrams
- Package Diagrams
- Agent-Message Diagrams

All these diagrams will be discussed in the followings. The first three kinds of diagrams are standard UML diagrams, that we suggest might be useful also for documenting EURACE software system. The last diagram is a newly proposed diagram, derived by a research activity aiming to find an effective way of graphically representing X-Agents and their interactions. Here we quote also an informal diagram used when designing X-Agents to graphically depict the dependencies among functions. In this diagram, that we might call Function Dependency Diagram, the functions to be called on different kinds of Agents are shown in bubbles, the time (or control) flows top to bottom, the dependencies between functions are shown with solid arrows, while the information produced by a function and used by another is shown with a dashed arrow starting from the former and pointing toward the latter. The functions that are executed in parallel are shown in horizontal regions, divided by red dotted lines. Before the functions in a region can be executed, the functions in the zone above it must in turn have terminated their execution. Figure 4, taken from Fig. 8 of Deliverable D1.1, shows one of such diagrams, representing the dependencies between Firm's and Household's functions in the labour market model. We decided not to formally define this kind of diagram, in this version of the graphical notation for X-Agent systems, but this might be accomplished in the future.

5.1 State Diagrams

State Diagrams have already been used to document the states and the transitions between states of an X-Agent. For instance, they are used in Deliverable D1.1. These diagrams represent the states of an object (an X-Agent in our case), and the transitions among them. In original UML State Diagrams, the states are represented by a rectangle with rounded borders; they may have a name, and may show the actions performed when the state is entered, during the time the object is in the state, and just before exiting from the state. Transitions are shown as solid arrows between a state and another. They are triggered by an event, whose name may be reported close to the arrow, together with preconditions. Applying State Diagrams to X-Agent states, we need just to name the states. The transitions between states are not due to events, but to the activation of a proper function, depending on possible preconditions on the Agents memory and on a message read by the function. This information is the same that is reported in the design tables described in chapter 5, and could be shown along the arrows representing state transition. However, in this way the State Diagram would become very clumsy, losing its ability to convey quickly the state and state transition information. Thus, we suggest to limit the UML State Diagrams to graphically show the states and the possible transitions among them, at most marking the transitions just with the name of the function able to trigger that transition. For instance, this is exactly how these diagrams are used in Deliverable D1.1. We report in Figure 5 the state diagram of a Firm, also shown in Fig. 3 of D1.1. Note that the black circle denotes the starting state of the Agent.

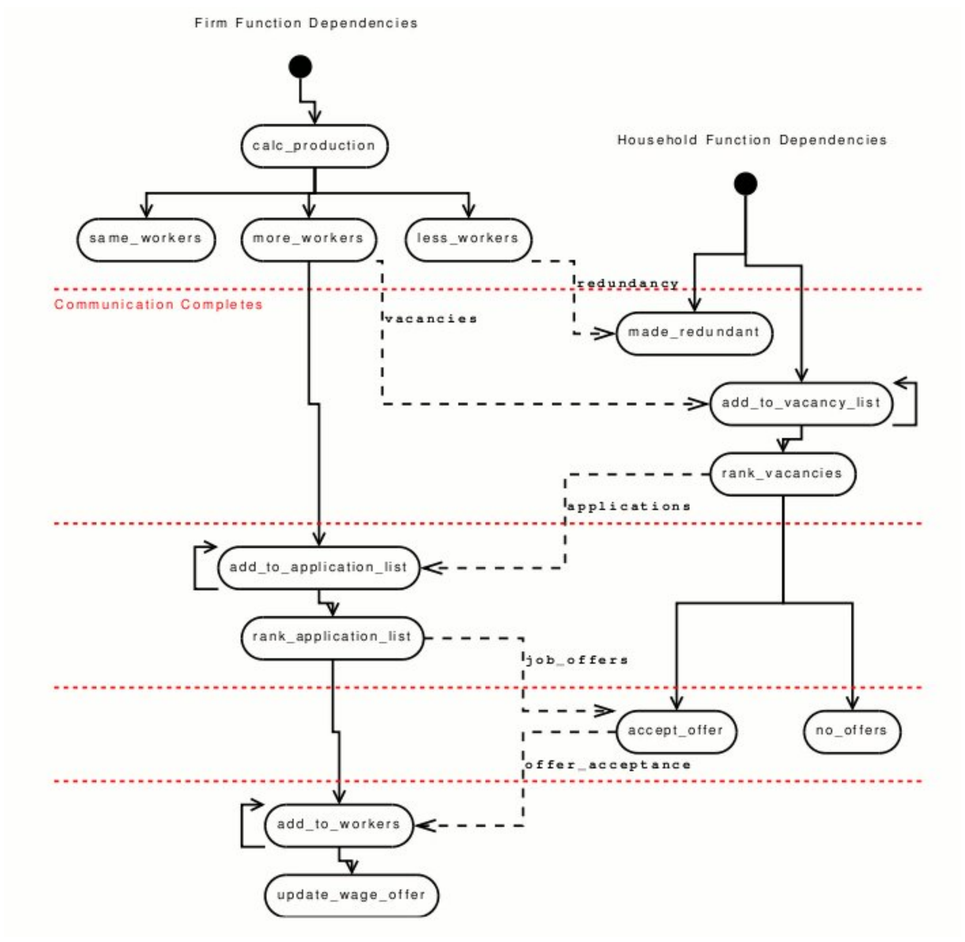


Figure 4: An informal diagram showing Function Dependencies for the labour market model.

5.2 Activity and Package Diagrams

In UML, Activity Diagrams are similar to the old flow diagrams used to document the control and information flow of a procedure. We believe that these diagrams are useful also to graphically document the key functions of EURACE model, when they are not simple enough to be documented with a comment. Clearly, only a small subset of the functions of the Agents in the system will likely need to be documented in this way. Writing diagrams for every trivial, or simple, function would be a waste of time. Figure 6 shows how a function is documented in a UML Activity Diagram.

If the system to model and simulate using X-Agents is large (as it is the case of EURACE), then it might be sensible to divide it into interacting subsystems, small enough to be documented and grasped with a reasonable

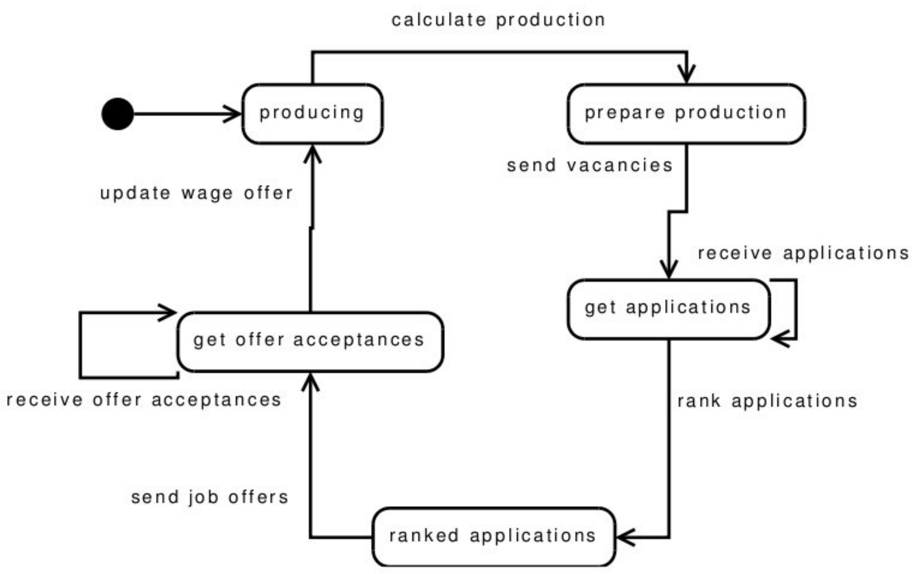


Figure 5: UML State Diagram applied to the states and the state transitions of a Firm

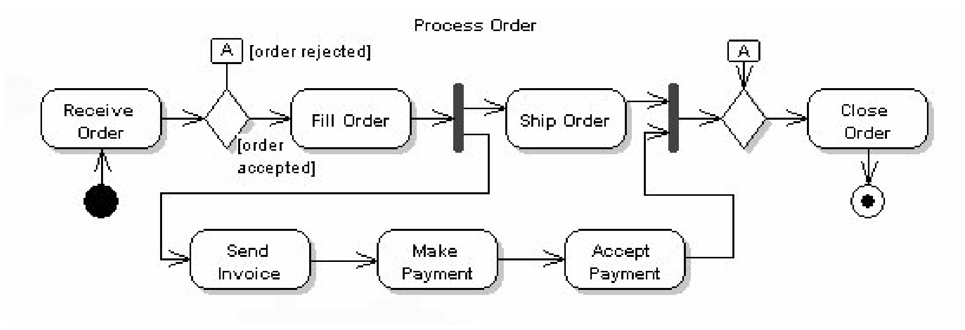


Figure 6: UML Activity Diagram used to document a function

effort. In UML, subsystems are called packages, and the UML Package Diagram shows the packages of a large system, and their dependencies. Figure 7 reports a very rough Package Diagram that could be a starting point to decompose a large economy model into packages, showing the notation of this kind of diagrams.

5.3 Agent-Message Diagrams

The aim of the Agent-Message Diagram (AMD) is to graphically represent the various kinds of X-Agents, and messages involved in a system or in a

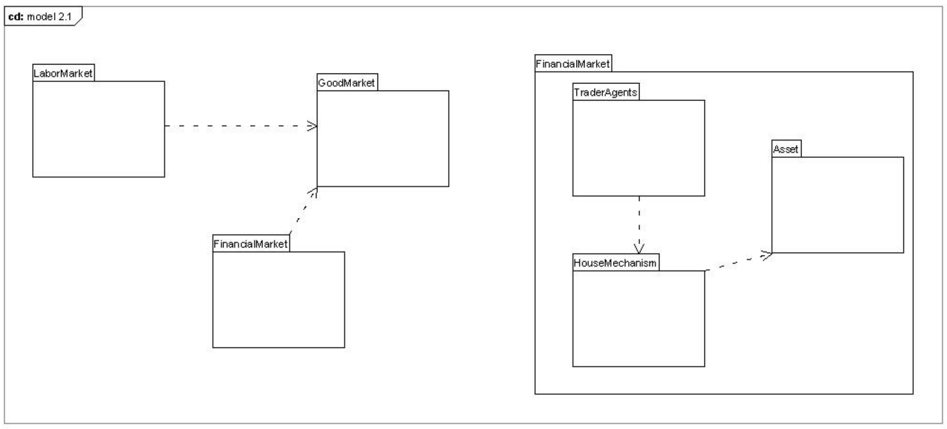


Figure 7: UML Package Diagram of a large economy simulator.

package, and the relationships among them. AMD are directly derived by UML class diagrams. In this first version (v. 1.0) they include a graphical representation of an X-Agent and of a message as nodes of the UML diagram, and two basic relationships: inheritance (between two agents) and message dependency (between an agent and a message).

5.3.1 X-Agent Representation

The graphical representation of an X-Agent is shown in Figure 8. As in UML class description, the X-Agent is represented by a rectangle divided into regions. In the upper region there is the name of the X-Agent, possibly preceded by the name of the package it is defined in (in the case of big models that are decomposed in packages, as in UML). In this case, the name of the package is followed by two colons and then by the Agent's name. In the example, we started the Agent's name with upper capital as in many UML diagrams, but this is not a constraint of the language. In UML class representation, the central region holds the attributes, that is the basic data hold by the class. In our Agent's representation, the central region holds the same information, that is what is contained in the memory of the Agent, including its state. This region is divided in two parts. The upper part shows the names and the types of the memory fields of the Agent. The lower part, divided by a dashed line, shows the states of the Agent. The initial state is highlighted by a black dot. The lower region, as in UML classes, shows the functions of the X-Agent. The functions directly invoked in the states and able to read and write external messages are prefixed by a "plus" sign ("+"). In UML, this symbol means "public access" to the function, which is a quite different concept. However, with X-Agents functions cannot be called by external entities, so the idea of "public" here is linked to the ability to access

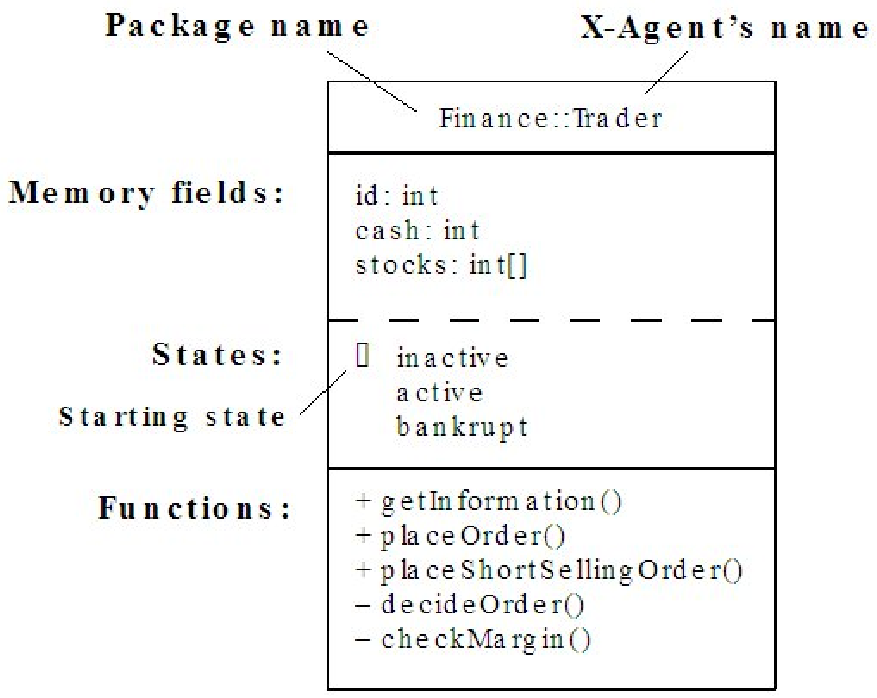


Figure 8: Graphical representation of an X-Agent.

publicy available information such as messages. The utility functions written to be called by other Agent's functions and able to access only its internal memory are marked with the "private access" UML symbol (a "minus" sign). Overall, the X-Agent representation is similar to the UML class one. This makes sense, because UML classes represent objects, that is entities with memory and operations (functions), exactly like X-Agents. The main difference between objects and agents regards their behavior and activation structure (synchronous, direct message sending with objects, state-driven asynchronous function invokation with X-Agents).

5.3.2 Message Representation

Figure 9 shows the message representation in Agent-Message Diagrams. X-Agent messages are simple record holding a proper information, with no function implemented by the message. Their AMD notation reflects this, and is in turn similar to that of UML classes, with two exceptions:

1. there is no lower region showing the operations;
2. there is an enveloped symbol before the name, to highlight it is a message.

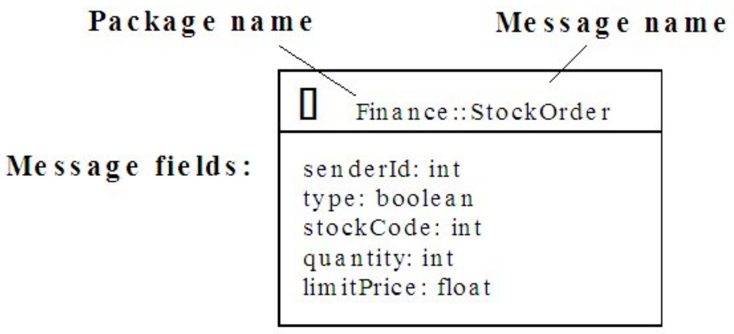


Figure 9: Graphical representation of a message.

As with X-Agents, the name can be prefixed by the package name. The region showing the data fields of the message is similar to analogous regions of class attributes and X-Agent memory, showing the names of the fields and their type.

5.3.3 AMD Relationships

We decided to have two kinds of relationships in AMD version 1.1: inheritance and message dependency. We decided to have two kinds of relationships in AMD version 1.1: inheritance and message dependency. Inheritance might be controversial at a first sight, because X-Agents do not use this paradigm. However, at analysis level, there might be different kinds of agents sharing common memory fields, states and functions. This happens very often in real economic systems, and we believe it should be highlighted in the analysis phase of system development. UML class diagrams, which AMD are derived from, have inheritance relationship, and we decided to use it. So, we will have base Agents and derived Agents, as we have superclasses and subclasses, respectively. When X-Agents are implemented, of course, the inheritance hierarchies present in the analysis will be flattened, and inherited data, states and functions will be duplicated in all derived agents. Inheritance relationship is directional, and is represented by a solid line with a triangular tip, as shown in Fig. 6. As regard message dependency, we know that the various kinds of X-Agents can write and read given kinds of messages. If an X-Agent can write a given message, a dependency arrow (a dashed line) will be drawn from the agent to the message. On the contrary, if an X-Agent can read a given message, a dependency arrow will be drawn from the message to the agent. If both things can happen, two arrows will be drawn. The notation for message dependency is shown in Figure 10.

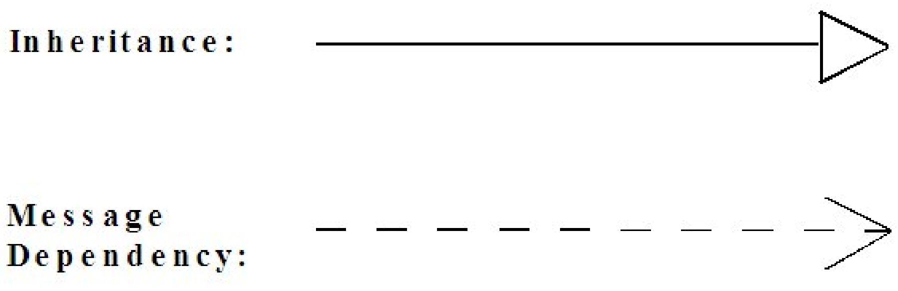


Figure 10: Graphical representation of inheritance and message dependency relationships.

5.3.4 Some Examples

In the remaining of this chapter, we will present some graphical examples of Agent-Message Diagram, according to the case studies presented in Deliverable D1.1. Figure 11 shows.

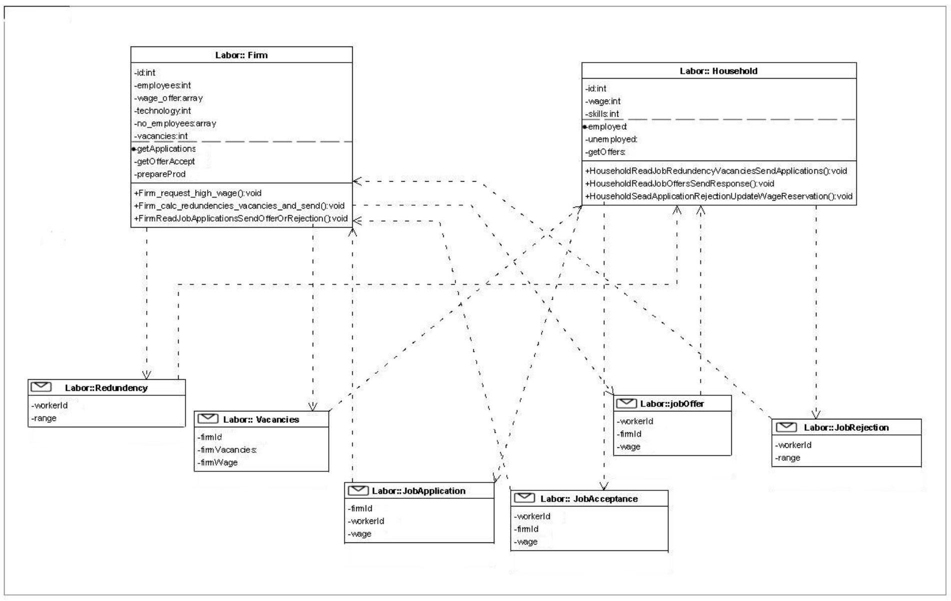


Figure 11: Agent-Message Diagram.

6 Testing Guidelines

6.1 Introduction

The traditional definition of "testing" comes from the world of quality assurance. We test software to find its bugs, being a bug a difference between existing and required behavior, state or quality. We continuously test software, because testing can prove that there are some bugs in it, but cannot prove that there is no bug. However, if no bug is found after thorough testing, we can be quite confident that the software is free of bugs, of at least of major bugs.

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. Software testing is a practice used for the *Verification* and *Validation* of the software.

During and after the development process, it's fundamental to verify the system to be assured that it satisfies the requested specifications and provides the functionalities expected by customers. The verification and validation activities are 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 guaranties 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?*".

Developing software is a complex business. The main goal of testing is to eliminate all bugs by going through the phases of requirement elicitation, analysis, design, implementation, integration and release. On the contrary, it is very inefficient to test the system only after its completion, concentrating all test efforts just before deployment, because testing is much more difficult, and because bugs are much more expensive to fix when the system is complete. The testing process of the software has two distinct main goals:

- Demonstrating to the developer and to customer that the software satisfies all requirements – if the software is customized, then there should be at least one test for each user requirement, if the software system is generic, then there should be tests for all the functionalities 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 his 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 this case, the test cases are planned to find the defects, can be deliberately obscure and don't need to reflect the general use of the system.

6.2 Types of test

There are mainly six distinct levels of the testing hierarchy:
Unit testing is the lowest level of testing. It is the testing of individual hardware or software units or groups of related units (*IEEE 1990*). Unit testing is any procedure used to validate the individual units of an application. For instance, in a procedural program it used to validate a function or a procedure, in object-oriented programming it is used to validate classes.
Integration testing is a kind of testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them (*IEEE 1990*). It follows unit testing and precedes system testing. It checks how independent pieces of code work together. The integration process of a system builds the system from basic components, so integration testing verifies how well the components work together.
Functional testing involves ensuring that the functionalities specified in the requirement specification work, while System testing is a testing conducted on a complete, integrated system to evaluate the system compliance with its specified requirements (*IEEE 1990*). *Functional* and *System testing* check the operation of all the subsystems working together, along with any interaction they have with their environment.
Acceptance testing is formal testing conducted to determine whether or not a system satisfied its acceptance criteria (the criteria the system must satisfy to be accepted by a customer) and to enable the customer to determine whether or not to accept the system (*IEEE 1990*).
Regression testing is selective retesting of a system or component to verify that modifications have not caused unintended effects and the system or

component still complies with its specified requirements (*IEEE 1990*).
Beta testing: when an advanced partial or full version of a software package is available, the development organization can offer it free to one or more (and sometimes thousands) potential users or beta testers.

Some combination of all these types of tests will help to deliver usable software systems.

It is worth noting that the basic kinds of tests are only two: unit tests, meant as a way to test the basic units of a program, and functional tests, intended to test complete functionalities. All other kinds of testing are a combination of these two kinds, or are the application of one of these kinds with a specific goal. For instance, integration testing is a combination of unit and functional testing to ensure that two or more modules developed separately work together; acceptance testing is functional testing specified by the customer and aiming to verify that the software satisfies its requirements; system testing is functional testing applied to the system as a whole; regression testing is a set of unit and functional tests applied after a system modification to verify system integrity.

In the last part of this section, we define in deeper detail unit testing and functional testing.

6.2.1 Unit Testing

functionality, the and it is features.

Unit Tests are pieces of code that are typically created by developers in order to prove that another, related piece of code does what the developer thinks it should do. For each software basic module she writes, the developer also writes tests that check whether it works correctly and efficiently. A unit is the smallest possible testable software component and various interpretations exist, depending on the software paradigm used:

- Procedure or function;
- Class or object or method;
- Small-sized component, for instance written in XML, HTML or SQL;

Unit Tests are 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 it works properly under every condition, or at least in the most likely conditions.
- With a bug occurs: first, a test that produces the bug is written, then the bug is eliminated. The test makes sure that bug is actually eliminated and that, in the future, it will not reoccur.

- When something is difficult to understand: if the interface is not clear or the implementation seems to be difficult or there are unusual arguments and side conditions required, a unit test helps to reduce the complexity.

Unit tests are a kind of "*white box*" (as opposed to "*black box*"), or "*glass box*" testing, because every detail of the item to test (the "box") is accessible by the tester.

While designing unit tests, we aim to check that the code does what we intended, so we aim to test very small, very isolated pieces of functionality. We verify that the functions generate the correct answers from our test data.

For instance, if we add a value to an ordered list, then we'll have to ensure that this value appears at the end of the list. Let us suppose to have a C-code function that sum two integers as shown in Fig. 12.

```
/* Function that sums 2 integers */
  int  addition(int a, int b)
  {
     return (a + b);
  }
```

Figure 12: Addition Function Example.

To verify that this function generates the correct answer, we might write its unit test as shown in Fig. 13.

```
  BOOL additionTest()
  {
    if(addition(2,3) != 5) return (FALSE);
    if(addition(0,0) != 0) return (FALSE);
    if(addition(6,0) != 6) return (FALSE);
    if(addition(-2,0) != -2) return (FALSE);
    if(addition(7,-2) != 5) return (FALSE);
    if(addition(-5,3) != -2) return (FALSE);
    if(addition(-5,-1) != -6) return (FALSE);
    return (TRUE);
  }
```

Figure 13: Addition Function: Unit Test Example.

Note that the unit test is broad, but cannot be in any way exhaustive. Of course, we cannot test all possible integer pairs, even constrained within the values compatible with computer precision! It is important, however, that the test includes cases that are intrinsically different, such as summing positive, negative and mixed values, and limit values, such as the zero in the example.

If the software system to test is written using a procedural language, as for instance C language, we test functions, and unit tests are in turn written as functions. If the system is developed using the object-oriented (OO) approach, with an OO language such as C++ or Java, unit testing is performed at the class level. For each class developed, there is a test class holding the methods able to test all significant methods of the original class.

In the case of X-machines and FLAME framework, the units to test are the X-machines themselves, considered as state machines, and the C functions implementing the system behaviour. Testing a state machine means to verify that state transitions are performed correctly from a given state to another. Since FLAME is written in C language, the unit tests are in turn C functions, possibly making use of global data defined and managed at a module level. More on this in next sections.

6.2.2 Functional Testing

Functional tests are pieces of code that are typically created by testers, who are developers specialized in writing and executing tests. On the contrary 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 linked to the technology used to develop the software system than unit tests. For instance, if the system to test is accessible through a Web interface, then it is possible to test it directly through this interface, irrespectively of the specific language and architecture of the system. FIT [Cun05] is a good example of automatic testing framework for performing functional tests on systems accessible through http protocol.

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 classes aimed to perform functional tests in the case of OO development, or writing test functions in the case of a procedural language.

predeterminate required

6.3 Agile Testing

The importance of testing in software development has been always stressed by software engineering. However, in traditional development testing was often performed only on the final system, and a substantial part of bug finding was left directly to the user, through beta testing and also to the final user. Moreover, most tests were executed by testers following guidelines and testing plans, but were performed stimulating the system "by hand“, resulting in a slow, costly and not perfectly repeatable manner.

Since finding and fixing bugs in a deployed system is very costly, these habits resulted in a great adversion to make changes in software systems. Changes were made only to fix bugs, or to add mandatory functionalities, and were strictly limited to the minimum. In this way, system "entropy“, or disorder tended to become bigger and bigger, making systems more and more difficult to modify and maintain.

This situation has been challenged by the advent of the so-called Agile Methodologies (AM) for software development. AM strive to be able to change system requirements in any phase of its development. This is accomplished through many different practices, applied together, like short iterations, continuous feedback, feature-driven development, continuous integration, refactoring and test driven development.

As regards testing, AM prescribe that every system must be endowed of an *automatic test suite*, able to thoroughly and repeatably test the system in a short time whenever it is needed. The availability of such an automatic test suite enables *refactoring.*

Based on many years of building and maintaining unit and functional tests, Meszaros [MSA03] introduced the *Test Automation Manifesto*, that gives key guidelines to create automated test. According to the Manifesto, automated test should be:

- **Concise**: Tests should be as simple as possible and no simpler.
- **Self Checking**: Tests should report their results in such a way that no human interpretation is necessary.
- **Repeatable**: Tests should be run repeatedly without human intervention.
- **Robust**: Tests should produce the same results now and forever. Tests should not be affected by changes in the external environment.
- **Sufficient**: Tests should verify all the requirements of the software being tested.
- **Necessary**: Everything in each test contributes to the specification of desired behavior.
- **Clear**: Every statement should be easy to understand.
- **Efficient**: Tests should run in a reasonable amount of time.

- **Specific**: Each test failure points to a specific piece of broken functionality.
- **Independent**: Each test should be run by itself, or in a suite with an arbitrary set of other tests, in any order.
- **Maintainable**: Tests should be easy to modify and extend.
- **Traceable**: Tests should be traceable to the requirements; requirements should be traceable to the tests.

6.3.1 Automated Testing

In AM, testing is of paramount importance in the whole development process of a software system. Willing to obtain the most advantage using tests, they have to be executed automatically.

In fact, automatic tests can be executed quickly whenever the developer feels they are necessary. This happens many times a day, even many times every hour of development. Automatic unit and functional tests are not only a way to test, but also help:

- **to perform system design**: when a test is difficult to write, usually it is a design problem, that can be found and fixed in the test writing phase;
- **to document the system**: a suite of well-written tests is a great documentation tool; functional tests may even be considered as executable specifications;
- **to easily change the code whenever needed**: easily running all tests against a change in the code ensures that code has not be broken, and no unwanted side-effects has been introduced;
- **to increase productivity**: the time and effort spent to write the automatic tests is more than repaid when the system becomes large, because a lot of traditional debugging and bug fixing is saved;
- **to make easier all other kinds of testing**: integration testing, regression testing and acceptance testing become straightforward if the system is endowed of the proper automatic test suite since the beginning;
- **to boost developer's confidence**: when are tests are passed, the developer feels that her work has been fine; both self-confidence and team's esteem are increased;

To write effective automatic tests, however, is not easy and requires method and planning. The lines of code devoted to testing are of the same order of magnitude of those of the system itself, and may even be of the same size. As said before, however, the effort spent in writing automatic tests is more than repaid in terms of system quality and ease of maintenance.

The most widely used automatic testing framework is X-Unit, first devised by Kent Beck for Smalltalk language (and dubbed "S-Unit"), and then ported to most other languages, including Java (J-Unit) and C (C-Unit). X-Unit and its C-unit implementation is described in section 6.4.

Many agile developers use the technique of "Test-Driven Development" (also known as "Test-Driven Design", to highlight the design aspect of testing).

6.3.2 Test-Driven Developement

Test-Driven Development (TDD) is a unit testing practice that is used by software developers as they write code. In TDD, all tests are written before the code to be tested, in bunches of testing and regular code as small as possible: first a little test is written, then a little code is added that enables the test to pass, and so on in this way.

Development proceeds through rapid iterations, applying following six steps:

1. Analyzing the project; identifying some objects and their properties that are fundamental for the project development;

2. Writing some automatic tests;

3. Running these unit test cases to ensure they fail, because there isn't the code to run yet;

4. Implementing code that should allow the unit test cases to pass;

5. Re-running the unit test cases to ensure they now pass with the new code;

6. Restructuring the production and the test code to make it run better and/or have a better design.

TDD that allows to improve the quality of the software, to reduce the cost of the testing phase, to reduce the number of faults that are due to the programmer during the testing phase, to help programmers to reorganize their code without breaking anything that they have already written.

TDD is an "extreme" technique, though very used when writing systems using an OO approach. When the system to be written is based on a procedural language such a C, however, its usage is much more difficult and questionable.

6.4 XUnit and C-Unit

XUnit is a general testing framework, first developed by Kent Beck for the Smalltalk language and then generalized and ported to many other languages. Here we will describe its main principles and patterns, describing its C implementation (C-Unit) in the next section.

Quoting from the original Beck's paper [Bec94], the philosophy of X-Unit is based on four key points:

- **Failures and Errors**: *"The framework distinguishes between failures and errors. A failure is an anticipated problem. When you write tests, you check for expected results. If you get a different answer, that is a failure. An error is more catastrophic, an error condition you didn't check for."*
- **Unit testing**: *"I recommend that developers write their own unit tests, one per class. The framework supports the writing of suites of tests, which can be attached to a class. I recommend that all classes respond to the message "testSuite", returning a suite containing the unit tests. I recommend that developers spend 25-50% of their time developing tests."*
- **Functional testing**[2]: *"I recommend that an independent tester write functional tests. Where should the functional tests go? The recent movement of user interface frameworks to better programmatic access provides one answer– drive the user interface, but do it with the tests."*
- **Running tests**: *"One final bit of philosophy. It is tempting to set up a bunch of test data, then run a bunch of tests, then clean up. In my experience, this always causes more problems that it is worth. Tests end up interacting with one another, and a failure in one test can prevent subsequent tests from running. The testing framework makes it easy to set up a common set of test data, but the data will be created and thrown away for each test. The potential performance problems with this approach shouldn't be a big deal because suites of tests can run unobserved."*

X-Unit is based on four key patterns, "Fixture", "Test Case", "Check" and "Test Suite", which we will describe in the followings.

- **Fixture**: in Beck's words: *"Testing is one of those impossible tasks. You'd like to be absolutely complete, so you can be sure the software will work. On the other hand, the number of possible states of your*

[2]In the original Beck's paper this is reported as "Integration testing", but in our definition of testing it is more proper to call it "Functional testing"

program is so large that you cannot possibly test all combinations. If you start with a vague idea of what you'll be testing, you'll never get started. Far better to start with a single configuration whose behavior is predictable. As you get more experience with your software, you will be able to add to the list of configurations. Such a configuration is called a "Fixture"."

So, a 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 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 system will have many Fixtures, each of which will be used by many tests, in many ways.

- **Test Case**: A test case is a single test, or an atomic set of tests, defined on a Fixture and implemented in (test) code, in such a way to be repeatable and transmissible to other developers.

  Using an OO language, a Test Case is implemented as a class, subclass of abstract class *TestCase*, having its methods the task to stimulate the Fixture(s) of the Test Case and check for the results.

  In a procedural language, a Test Case is implemented as a set of (test) functions, operating on common variables holding the Fixture.

  It is important that the Test Case is endowed of a function or method (usually called "*setUp()*") that creates or reinitialize the Fixture, and possibly of a function (usually called "*tearDown()*") that releases whatever resources used for the test.

- **Check**: 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. 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(*num expression1, num 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. In the case of dynamic languages such as Smalltalk and Ruby, even catastrophic errors – those errors that would abort the program when occurring at runtime – can be dealt with, recorded and testing skipped to the next Text Case. If the language support exceptions, Test Cases must be run in this mode, and unanticipated errors can be dealt with (at least to some extent) using the exception mechanism.

- **Test Suite**: Test Cases are grouped and organized in Test Suites, sets of Test Cases that can be run automatically and repeatably. 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, recursively, 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.

When using X-Unit *Framework* to write Unit Tests, the typical sequence of steps is the following.

1. Define a class *MyTest* that is subclass of *TestCase*. *TestCase* is a class contained in the library X-Unit (such as JUnit or SUnit).

2. Override the *setUp*() method to initialize the variables or resources under test.

   These variables are called *Fixture*.

3. Override the *tearDown*() method to release the variables and resources under test. Each test runs in the context of its own fixture; calling *setUp*() before and *tearDown*() after each test ensures there are no interferences or side effects among test runs. So, the execution of an individual unit test proceeds as follows:

   ```
   setUp(); .. /* Body of test */ .. tearDown();
   ```

4. Define one or more methods *testXXX*() that exercises the unit under test and asserts expected results. The test methods must be named starting with "Test".

5. Define Assertions. An Assertion is a function that verifies the behavior of the unit under test. Failure of an assertion typically throws an exception, aborting the execution of the current test.

6. Define a *main*() method that runs the Test Case in batch mode. Running a TestCase automatically invokes all of its public *testXXX*() methods.

7. Optionally define a static *suite*() method that creates a TestSuite containing all *testXXX*() methods of the Test Case. A Test Suite is a set of tests that all share the same Fixture.

   All Test collected in a Test Suite should be able to be executed one by one, or all together. This functionality is very useful when it is necessary to verify the functionalities of the whole system.

X-Unit framework is very powerful, yet very simple and easy to use. Martin Fowler, speaking about J-Unit, the implementation of X-Unit for Java language written by Kent Beck and Eric Gamma, affirms that:
"Never in the field of software development was so much owed by so many to so few lines of code"

6.4.1 C-Unit

C-Unit is a framework for writing and running unit tests in C code. It follows the principles and patterns of X-Unit, though adapted to a procedural language such a C. C-Unit provides a convenient way of writing and executing unit tests, allowing to implement test cases, test suites, assertions, etc.

However, unlike other testing frameworks for more dynamic languages, such as S-Unit for Smalltalk and J-Unit for Java, C-Unit is not able to intercept runtime errors, because it is not provided of language constructs to manage exceptions. In the case of errors resulting in the testing session abort, the developers need to start a debugging session using the debugging tools of the specific operating system and C compiler or IDE.

C-Unit uses a simple framework for building test structures, and provides a rich set of assertions for testing common data types. The success or failure of these assertions is tracked by the framework, and can be viewed when a test run is complete. Each assertion tests a single logical condition, and fails if the condition evaluates to *FALSE*.

Some common assertions defined by C-Unit are:

- CU_ASSERT_TRUE(value)
  Assert that value is TRUE (non-zero)

- CU_ASSERT(int expression)
  Assert that expression is TRUE (non-zero)

- CU_ASSERT_FALSE(value)
  Assert that value is FALSE (zero)
- CU_ASSERT_EQUAL(actual, expected)
  Assert that actual = = expected
- CU_ASSERT_DOUBLE_EQUAL(actual, expected, granularity)
  Assert that |actual - expected| ≤ |granularity|; Math library must be linked in for this assertion

For instance, we can shown a unit test of a function that returns the maximum of two integers, the algorithm using C-Unit is following:

```
/* Function that returns the maximum of 2 integers */
   int maxi(int a, int b)
   {
      return (a > b) ? a : b;
   }

/* C-Unit Test of the maxi() function */
   void test_maxi(void)
   {
     CU_ASSERT(maxi(0,2) == 2);
     CU_ASSERT(maxi(0,-2) == 0);
     CU_ASSERT(maxi(2,2) == 2);
   }
```

CUnit is organized like a conventional unit testing framework. Individual test cases are packaged into suites, which are registered with the active test registry. Suites can have setup and teardown functions which are automatically called before and after running the suite's tests. All suites/tests in the registry may be run using a single function call, or selected suites or tests can be run. The test registry is the repository of suites and associated tests. CUnit maintains an active test registry which is updated when the user adds a suite or a test. The suites in this active registry are the ones run when the user chooses to run all tests.

The CUnit test registry is a data structure `CU_TestRegistry` declared in `<CUnit/TestDB.h>` It includes fields for the total numbers of suites and tests stored in the registry, as well as a pointer to the head of the linked list of registered suites. The user normally only needs to initialize the registry before use and clean up afterwards.

The functions that allows to initialize and clean the registry are respectively `CU_initialize_registry()` and `CU_cleanup_registry(void)`. In order to run a test case, we must add it to a suite, registered with the test registry. The function `CU_add_suite` allows to add a suite to the registry, while a test case is added to a suite using the function `CU_add_test`. In the case of large test structures, with many tests and suites, managing test/suite associations and registration is complex. CUnit provides a special registration system helping to manage suites and tests, so it is possible to group test cases into an array of `CU_TestInfo` instances, defined as in the following:

```
CU_TestInfo test_array1[] = {
  { "testname1", test_func1 },
  { "testname2", test_func2 },
  { "testname3", test_func3 },
  CU_TEST_INFO_NULL,
}; E
```

Each function name in the array is unique, and corresponds to a single test case. The test cases included in a single `CU_TestInfo` array form the set of tests that can be registered with a single test suite [AK04].

Suite information is defined using one or more arrays of `CU_SuiteInfo` instances :

```
CU_SuiteInfo suites[] = {
  {"suitename1", suite1_init-func, suite1_cleanup_func, test_array1},
  {"suitename2", suite2_init-func, suite2_cleanup_func, test_array2},
  CU_SUITE_INFO_NULL,
};
```

Using a procedural language such as C, a best practice to design large software systems is to use separated modules, each included in a single file. Each module represents an encapsulated data type, and is constituted by two files: *module.c* holding the source code, and *module.h* holding the interface information.

The interface includes all data structures and the signatures of the public functions available to be called by other modules. When a module has to be used by another module, its interface is included in the latter module, and it is possible to declare variables belonging the the types defined in the included module.

To define a test suite, we need to define a test module whose name should be provided of "Test" prefix, followed by the name of the module to be tested. This test module is in turn an encapsulated module and hold the test suite associated to the module to be tested. In this suite we have to define variables belonging to the module to be tested, and to create set-up functions able to initialize them to well defined and significant values. A set of these variables, properly initialized, is a Fixture.

It is possible to define many set-up functions, each of which is used to configure a different fixture. These functions are called from within test cases, and might allocate external resources, such as file streams, sockets, database connections. When the test case ends, we need to release these resources, and this is accomplished by a clean-up function called in the end of the test case, tipically called `<TestCase>_clean()`, where `<testCase>` is the name of the test case.

The module implementing a specific test suite must provide in its interface the signature of test functions and the array of type `CU_TestInfo` holding the suite itself. Following the same modular approach of defining modules composed by other, lower level modules, test suites are included in other higher level suites, and registered. It is therefore convenient to define a separate header module, representing the test hierarchy.

Once the suite collection is defined, we have to run the tests. Consequently, an executable file (a `main()`) is needed, able to run the test cases, and to produce their reports.

CUnit provides native functions for running suites and tests. During each run, the framework keeps track of the number of suites, tests, and assertions run, both passed and failed. The modules *Automated*, *Basic*, and *Console* are included in the CUnit library and allow to run all tests in all registered suites. The automated and basic interfaces are non-interactive. The basic mode uses the standard output to print out the results, while the results of tests ran in automated mode are output to an XML file.

The console interface enables interactive mode to run a registered suite. Users can select the suite and run the tests interactively. The console prints the results to the standard output. To start a console session, use the function `CU_console_run_tests`.

When testing a medium or large software system with C-Unit, we need to define a module (named for instance "*suites*") holding the test suites to register, and an executable able to launch them (named for instance "*launcherTest*". The *suites* module shall hold an array of type `CU_SuiteInfo` whose entries are the various suites defined for testing. The executable shall use one of the running libraries, for instance the console. The *suites* module is responsible for adding test cases to the registry, while the launcher is responsible for initializing and cleaning-up the registry, and for executing the test suites chosen by the user.

In order to test a module with the C-Unit framework, the following sequence of steps should be made:

1. Create a new file having the name TestModule
2. Write the function to initialize the suite having the signature
   `void suite_modulename_init(void);`
3. Write some fixtures' functions to initialize the state of the encapsulated type or module variable in general
4. Write functions for tests having the signature: `void test_func(void);`

5. Define into the TestModule.h an array of type `CU_TestInfo` that contains all test case

6. Write the function to clean the suite having the signature
   `void suite_modulename_clean(void);` in order to deallocate the globals resources

7. Add the Test Suite to the test registry, writing a new item into the array suites

6.4.2 Unit-Test for an ADTs

Another, more realistic example is shown in the followings. Let us consider the Abstract Data Type (ADT) *Random*, that should be used in FLAME. It is an encapsulated type of data that allows to generate random values, using some procedures that handle data[3]. For instance, there are procedures returning known distribution functions such as the *gaussian()* distribution that answers a normally distributed random number with zero mean and standard deviation 1.0, or *nextBetween()* distribution that answers a random number uniformly distributed between a given minimum and maximum value, and so on. We created a suite implementation module, called "Suites", and a LauncherTest modul. In the module Suites (this module is a couple of file `Suites.c` and `Suites.h`) we implement a function that adds the test functions to the suite and the suites to the registries. The LauncherTest is used to launch the automatic tests using a console. In order to create a suite of ADT Random we can proceed step by step in the following way:

1. Create a file.c called Random.c.
   In this file we implement the ADT Random, provided of a data structure and some procedures, for instance the distribution functions cited above (see the ADT Random Implementation).

2. Create a file.h called Random.h
   This file contains the interface of ADT Random implemented in the file Random.c (see the ADT Random Interface).

3. Create a file.c called TestRandom.c;
   In this file we implement the test functions of all distribution functions defined in the file Random.c; a given suite is initialized and is cleared up (see the TestRandom implementation);

4. Create a file.h called TestRandom.h;
   This file contains the interfaces of functions to init and to cleanup the

[3]ADT Random is based on Park-Miller algorithm [PSK88]. It uses two numeric constants, equal to 2147483647 and 16807, two derived constants equal to the integer quotient and remainder of dividing the former by the latter constant, and the seed of the algorithm, s.

suite and of the function that adds the test functions to the suite (see the TestRandom Interface).

The following code represents an implementation of the random number generator. It has some constants, a structure holding the data needed for this random number generator – in our case only a field called `seed` of type `double`, and a set of functions able to manipulate the data structure and generate a random number of a given type, based on the basic random number generator function, `next()`, that returns a random `double` number uniformly distributed between 0 and 1.

```
/* File "Random.c" :
 ADT Random Implementation-- EURACE Project
 Encapsulated type that allows to generate random values.
 Instance Variables:
    seed:  the random algorithm seed
*/
  # include ''Random.h''
  #define Rand_a  ((double) 16807)
  #define Rand_m  ((double) 2147483647)
  #define Rand_q = quo(Rand_m ,Rand_a);
  #define Rand_r = fmod(Rand_m ,Rand_a);
  #include ''stdio.h''

  /* Create a new instance of ADT Random */

  Random *newRandom()
  {
    Random *temp;
    temp = malloc(sizeof(Random));
    return temp;
  }

  /* Initialize: Set a reasonable Park-Miller starting seed */

  void initializeRandom(Random *aRandom)
  {
    aRandom->seed =(double)clock()/((double)CLOCKS_PER_SEC / 1000);
    next(aRandom);
  }

  /* Accessing the structure */

  double getSeed(Random *aRandom)
  {
    return aRandom->seed;
  }
  void setSeed(Random *aRandom, double seed)
  {
    aRandom->seed = seed;
  }

  /* The next function generates random instances of
     Float uniformly distributed in the interval 0 to 1 */

  double next(Random *aRandom)
  {
    double answer;
    aRandom -> seed = nextValue(aRandom);
    answer = aRandom->seed/Rand_m;
    return answer;
  }

/* The nextBetween() function generates random instances of Float
   uniformly distributed in the interval double min to double max
*/

 double nextBetween(Random *aRandom, double min , double max)
 {
   double answer;
   answer= min +  next(aRandom) * (max - min);
   return answer;
 }
 /* The gaussian() function generates random instances of Float
    normally distributed with zero mean and standard deviation
     equal to 1 */

   double gaussian(Random *aRandom)
 {
   double gauss;
   int k;
   gauss = -6;
   for (k=0;k<12;k++)
     gauss = gauss + next(aRandom);
   return gauss;
 }
```

The following code represents the header file *Random.h*, holding the interface of the random number generator. Note that the actual structure `Random` is defined in this file, that is included in the previous one.

```
/*  File "Random.h" :
 ADT Random Interface-- EURACE Project
 Encapsulated type that allows to generate random values.
 Instance Variables:
    seed  the random algorithm seed
*/
```

```
#include <time.h>
#include <math.h>
#include <stdlib.h>

  #define Rand_a  ((double) 16807)
  #define Rand_m  ((double) 2147483647)
  #define Rand_q = quo(Rand_m ,Rand_a);
  #define Rand_r = fmod(Rand_m ,Rand_a);

typedef struct Random Random;

struct Random {
      double seed;
};
//initialie: Set a reasonable Park-Miller starting seed
void initializeRandom(Random *aRandom);

// create an instance of Random
Random *newRandom();

//accessing at the structure
double getSeed(Random *aRandom);

void setSeed(Random *aRandom, double seed);

//private methods
double nextValue(Random *aRandom);


//Answer a normally distributed random number
with zero mean and standard deviation 1.0.
double gaussian(Random *aRandom);
```

Our implementation needs the function `quo(a,b)` which, given two double precision numbers, returns the integer quotient of the ratio a/b. This function is defined below:

```
/* File "quo.c" :
ADT Double Implementation-- EURACE Project
Encapsulated type that allows
to extend the double number functionality.
*/

#include "Double.h"

/*quo is a function that extends the
functionalities of the Double ADT.
It answers the integer quotient*/
```

```
 double quo(double c,double b) {
    double temp;
    temp= (int)(c/b);
    return temp;
  }
```

The interface of the test case written to test the random number generator and its functions is reported in the code below. At its end, it includes the definition of the `CU_TestInfo` array that holds the references to the test functions to be called to test the random number generator.

```
/* File "TestRandom.h" :
 TestRandom Interface--
 Test Module Interface for the ADT Random
-- EURACE Project
*/
#include "CUnit\Basic.h"
#include "CUnit\CUnit.h"
#include "Random.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#define TESTGAUSS 1.8442632806693

/* init and clean-up suite */
 int suite_random_init(void);
 int suite_random_clean(void);

/* test case for next */
 void testnext(void);

/* test case for nextBetween */
 void testNextBetween(void);

/* test case for  gaussian */
 void testGaussian(void);

/* Instance of CU_TestInfo array that allows
 to test large software systems */
 static CU_TestInfo tests_random[] = {
  { "test next", testnext },
  { "test nextBetween", testNextBetween },
  { "test gaussian", testGaussian },

    CU_TEST_INFO_NULL,
};
```

The C code that actually implements the interface of the test case for the random number generator is shown below:

```
/* File "TestRandom.c" :
 TestRandom implementation--
 Test Module Implementation for the ADT
Random -- EURACE Project*/ #include "TestRandom.h"

/*instance of ADT Random*/
Random *rng;

/*set the Fixture to test the random generating functions */

 static void setFixture(Random *rng)
{
    double co;
    co = (double) 2345678901.0;
    setSeed(rng, co);

}

/* init and clean-up suite */
 int suite_random_init(void) {
 rng=newRandom();
 setFixture(rng);
 return 0;}

 int suite_random_clean(void)
  { free(rng);
  return 0; }

static double  randomValues[]={
    0.149243269650845,0.331633021743797,0.75619644800024 ,
    0.393701540023881, 0.941783181364547, 0.549929193942775,
    0.659962596213428 , 0.991354559078512, 0.696074432551896,
    0.922987899707159};

/*test case for next*/
void testnext() {
    int k;
    double result[10];
    setFixture(rng);
    for(k=0;k< 10;k++) result[k]= next(rng);
    for(k=0;k< 10;k++)
    CU_ASSERT_DOUBLE_EQUAL(randomValues[k], result[k], 0.000001);

}

/*test case for nextBetween*/
void testNextBetween() {
     double temp;
     setFixture(rng);
     temp=nextBetween(rng, 1, 2);
     CU_ASSERT_DOUBLE_EQUAL(temp, 1.149243269650845, 0.000001);
   ;
}

/*test case for  gaussian*/
void testGaussian() {
     double temp;
     setFixture(rng);
     temp=gaussian(rng);
     CU_ASSERT_DOUBLE_EQUAL(temp, TESTGAUSS, 0.000001);
}
```

The following code represents the implementation of the test suite. In practice, it includes a single function, `AddTests()`, that adds all the tests of the test case to the registry of the suite.

```
/* File "Suites.c" :
 Suites Implementation--
 Module to manage and register the suites-- EURACE Project
*/
#include "Suites.h"

//add all test to the suites
void AddTests() {
  assert(NULL != CU_get_registry());
  assert(!CU_is_test_running());
    /* Register suites. */
    if (CU_register_suites(suites) != CUE_SUCCESS) {
        fprintf(stderr, "suite registration failed - %s\n",
            CU_get_error_msg());
        exit(EXIT_FAILURE);
    }

}

/*Suite Module-- Module interface to manage and register the suites
-- EURACE Project*/

#include "TestRandom.h"
#include "TestFixedCollection.h"

//add all test to the suites
void AddTests();

 static CU_SuiteInfo suites[] = {
  { "suite_random_both",  suite_random_init, suite_random_clean,
   tests_random },
  { "suite_FixedCollection_both",  suite_FixedCollection_init,
   suite_FixedCollection_clean, tests_FixedCollection },

    CU_SUITE_INFO_NULL,
};
```

In the end, we report the code of the main program written to run the tests of the suite.

```
/* File "LauncherTest.c" :
 LauncherTest Module,--EURACE project
 main module to run the tests and print the results of
 assertions
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include
"Suites.h"
#include "CUnit\Console.h"

int main(int argc, char* argv[]) {
  CU_BOOL Run = CU_FALSE ;

  setvbuf(stdout, NULL, _IONBF, 0);

  if (argc > 1) {
    if (!strcmp("-i", argv[1])) {
      Run = CU_TRUE ;
      CU_set_error_action(CUEA_IGNORE);
    }
    else if (!strcmp("-f", argv[1])) {
      Run = CU_TRUE ;
      CU_set_error_action(CUEA_FAIL);
    }
    else if (!strcmp("-A", argv[1])) {
      Run = CU_TRUE ;
      CU_set_error_action(CUEA_ABORT);
    }
    else if (!strcmp("-e", argv[1])) {
      print_example_results();
    }
    else {
      printf("\nUsage:
        ConsoleTest [option]\n\n"
        "Options:
```

```
                 -i   Run, ignoring framework errors [default].\n"
       "         -f   Run, failing on framework error.\n"
       "         -A   run, aborting on framework error.\n"
       "         -e   Print expected test results and exit.\n"
       "         -h   Print this message.\n\n");
    }
  }
  else {
    Run = CU_TRUE;
    CU_set_error_action(CUEA_IGNORE);
  }

  if (CU_TRUE == Run) {
    if (CU_initialize_registry()) {
      printf("\nInitialization of Test Registry failed.");
    }
    else {
      AddTests( suites);
      CU_console_run_tests();
      CU_cleanup_registry();
    }
  }

  return 0;
}
```

6.5 Unit tests in FLAME

FLAME is a framework for defining and running systems based on X-Machines. Since FLAME is based on C language, the testing framework of choice is C-Unit. In this section, we will give some guidelines for designing and writing unit tests for FLAME.

In FLAME, each X-Machine agent is defined giving:

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.
5. C code: the global data definition, the source code of the functions implementing the messages, and any other utility function used in the project, properly divided in files.

X-Machine agents are defined using XMML – X-Machine Agent Markup Language – based on XML. As regards testing, we deem that by now it would be premature to write a specific FLAME testing framework able to directly use the XMML definitions to help generating tests, so testing should be performed by writing test code in C, using C-Unit framework.

To this purpose, the guidelines to be used for designing and writing tests are the following:

1. Define tests for the common ADTs and general service functions used.
2. For each X-Machine:
   (a) define one or more Fixtures representing the X-Machines to be used in the tests;
   (b) define and code the tests to check for correct transitions between its states;
   (c) define and code the tests to check functions implementing the messages;
   (d) define and code other tests that might be relevant to the X-Machine.
3. Define the Fixtures to test for interaction among different X-Machines.
4. Define and code the corresponding tests.
5. Define and code the Test Suites to group the Test Cases defined.

6.5.1 An Example: The Financial Market

As an example of FLAME testing, we implemented in FLAME a simple financial market model, which we describe thereof. This simple model describes and simulates a stock market populated by two kinds of agents, that is traders and a clearing house. The traders' behaviour implemented is the simplest possible– traders issue buy/sell orders on the market randomly. For this reason, we call them "random traders". Random traders can trade only one asset. They have finite cash and stock amounts and issue buy/sell limit orders constrained by their budget. In the artificial stock market there is only one stock, traded in exchange for cash; the stock pays no dividend, there are no transaction costs or taxes, and the cash does not earn an interest rate. The prices are computed by a clearing house, an agent that has the responsibility to take the orders as inputs, to clear the price of the asset and finally to execute the orders that match with the new price. At each simulation step, each trader decides whether to trade or not, with a random extraction with given probability (set to 10%). All the orders are sent to the clearing house, that computes the new price. The order issued by a generic trader is a limit order, that is an order with a limit price. An order can be a buy or a sell order. If it is a buy order, the limit price is the maximum value that a trader wishes to spend. If an order is a sell order, the limit price is a minimum value that a trader wishes to get for selling. The random traders chooses to buy or sell at random, with equal probability.

The trader executes the following main actions:

1. she chooses whether to trade or not;

2. she chooses whether to place a buy or a sell order;

3. she sets the quantity and the limit price; these values depend on the trader's budget;

4. after price clearing, she gets the information about order execution by the clearing house

The clearing house executes the following main actions:

1. it receives the orders by the traders and collect separately buy orders and sell orders;

2. it determines the clearing price of the asset by crossing the demand and supply curves given by the current limit orders, stored in a collection of orders. This action is executed sending a request to the clearing mechanism;

3. it executes the limit orders matching with the new price

Each trader has a portfolio. The portfolio is an abstract data type with the responsibility to manage the resources of the trader herself, that is the cash and the quantity of stock owned.

A trader cannot place a buy order for more stocks than she can buy with her current cash. In the same way, she cannot sell more stocks than those currently owned.

The random trader implemented in this preliminary model has no intelligent strategy, because she issues orders randomly. She issues on the market buy or sell limit order, with the same probability p = 50%.

As stated before, a trader contains two abstract data types: Portfolio and Order. The behavior of a trader is implemented considering a module called Trader, which contains a file representing the interface of all functions that rule the behavior of the trader herself and a file represented the implementation of the functions; these files are called respectively *Trader.h* and *Trader.c*.

The clearing house computes the new price of the asset based on a predefined pricing mechanism, that might be a continuous double auction (also

Table 1: State transition table for *Trader* X-Agent

Start State	Message Input	Function	End State	Message Output
inactive		selectStrategy	waiting, inactive	order
waiting	orderStatus	updateTrader	inactive	

Table 2: State transition table for *ClearingHouse* X-Agent

Start State	Message Input	Function	End State	Message Output
receiveOrder	order	receiveOrder	computeAndIssue	
computeAndIssue		computeAssetPrice	receiveOrder	orderStatus

known as "book"), or a clearing mechanism as described before. So, the best solution is to delegate the responsibility of the computation of the asset price to another module. In this preliminary model, we used the clearing mechanism to clear the price; this has been implemented in a module called *clearingMechanism.* So, we developed also two files: *ClearingHouse.h* and *ClearingHouse.c*, that are respectively the files containing the interface and the implementation code of all functions that implement this pricing mechanism.

6.5.2 FLAME Implementation of the Financial Market

Our FLAME implementation of the financial market has two kinds of X-Agents: Trader and ClearingHouse, implementing the behavior described before.

There are two kinds of messages that are used by these agents:

- Orders, written by traders who place orders.
- OrderStatus, used by the ClearingHouse to notify the traders whether their orders have been executed or not.

The Tables 1 and 2 shows the pre and post state interactions of the trader agents.

Here a *Trader* is characterized by two states: *inactive*, when he is waiting to decide whether to trade or not, and possibly to place an order, and *waiting*, when he placed a limit order and is waiting to see the cleared price and whether the order hase been executed or not.

Every trader in inactive state executes the *selectStrategy* function, but only those traders who decide to trade (this happens with 10% probability) place and order as output and make a transition to the *wait* state. All other traders remain in the *inactive* state.

Table 3:

```
< datatype >

< name > Order < /name >

< var >< type > double < /type >< name > limitPrice < /name >< /var >
< var >< type > int < /type >< name > quantity < /name >< /var >
< var >< type > int < /type >< name > type < /name >< /var >

< datatype >
```

Traders in the *wait* state, when the price has been cleared and the *orderStatus* messages have been sent by the *ClearingHouse* agent, execute the *updateTrader* function, reading the status of their order, changing their internal memory accordingly and making a transition to the *inactive* state.

The *ClearingHouse* is a unique agent of the system, which at each step reads all the *order* messages, clears the price balancing sell and buy orders, decides which orders are executed and outputs the *orderStatus* messages accordingly. Its states are *recevingOrders*, when it is waiting for all orders to be submitted by the active traders, and *computeAndIssue*, when it processes the orders, posts the results and go back to the former state.

Order processing is performed by the *receiveOrder* function, while price clearing and posting the results for each order is performed by *computeAssetPrice* function.

The proper sinchronization of the model is accomplished by setting the *receiveOrder* function to be executed after *selectStrategy* function, and *updateTrader* function to be executed after *computeAssetPrice*, as is shown in Fig. 14

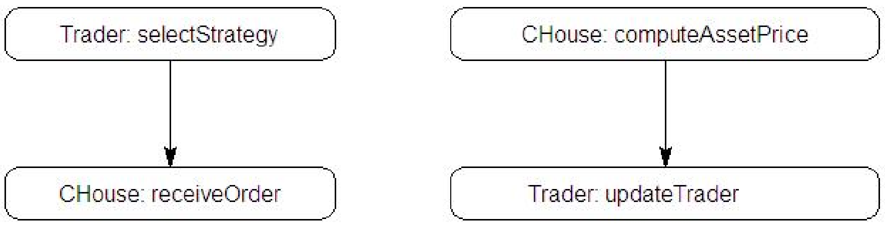


Figure 14: Function dependency of the Financial Market

An Agent-Message Diagram of this simple model is shown in Fig. 15.

The data representing the traders' orders and the portfolios are implemented using abstract data types. Their XXML definition is reported in Tables 3 and 4 respectively

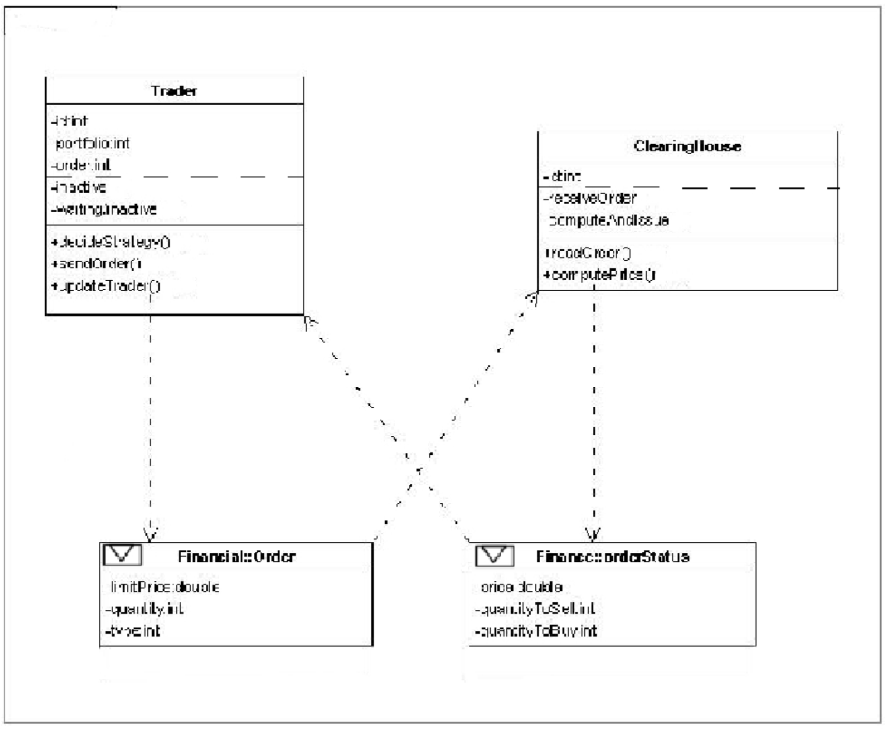


Figure 15: Agent-Message Diagram of the Financial Market

The XXML definitions of X-Agents Trader and ClearingHouse are reported in Tables 5 and 6 respectively

The clearing house computes the new price by cumulating buy and sell orders, using two collections of orders. Since this is a complex data structure, that would not easily be placed in an X-Agent memory, and since it is a singleton, that is an entity whose there is only one instance in the system, we implemented the clearing mechanism as a set of global data, accessed by global functions. These global data include two collections of orders, and a collection holding all past prices, which new prices are added to.

Though the presented market model is highly stylized, it is still able to produce a price series with some realistic features. Fig. 16 shows the price series of a typical simulation with 400 agents, lasting 5000 iterations, each representing a trading day. Fig. 17 shows the daily returns of the same price series, exhibiting non-Gaussian behaviour.

6.5.3 Testing ADT and auxiliary functions

In order to test the system, is necessary to test each module in the system, including the data and functions working as helpers of the X-Agent

Table 4:

```
< datatype >

< name > Portfolio < /name >

< var >< type > double < /type >< name > cash < /name >< /var >
< var >< type > int < /type >< name > quantity < /name >< /var >
< var >< type > double < /type >< name > usedCash < /name >< /var >
< var >< type > int < /type >< name > usedQuantity < /name >< /var >

< datatype >
```

Table 5:

```
<! − − − − − −XMachineAgent − Trader − − − − − − −− >

< xmachine >
< name > Trader < /name >

<! − − − − − −Variables − − − − − − −− >
< memory >
< var >< type > int < /type >< name > id < /name >< /var >
< var >< type > int < /type >< name > tau < /name >< /var >
< var >< type > double < /type >< name > strategy < /name >< /var >
< var >< type > double < /type >< name > wealth < /name >< /var >
< var >< type > double < /type >< name > posx < /name >< /var >
< var >< type > double < /type >< name > posy < /name >< /var >
< var >< type > Portfolio < /type >< name > portfolio < /name ><
/var >
< var >< type > Order < /type >< name > order < /name >< /var >
< memory >
<! − − − − − −Definingfunctions − − − − − − −− >
< functions >
< function >< name > sendOrder < /name >< /function >
< function >< name > selectStrategy < /name >< /function >
< function >< name > updateTrader < /name >< /function >
< /functions >
< /xmachine >
```

Table 6:

```
<! − − − − − −XMachineAgent − ClearingHouse − − − − − − −− >

< xmachine >
< name > ClearingHouse < /name >

<! − − − − − −Variables − − − − − − −− >
< memory >
< var >< type > int < /type >< name > lastPriceValue < /name ><
/var >
< var >< type > int < /type >< name > lastVolumeValue < /name ><
/var >
< memory >
<! − − − − − −Definingfunctions − − − − − − −− >
< /functions >
< function >< name > receiveOrder < /name >< /function >
< function >< name > computeAssetPrice < /name >< /function >
< /functions >
< /xmachine >
```

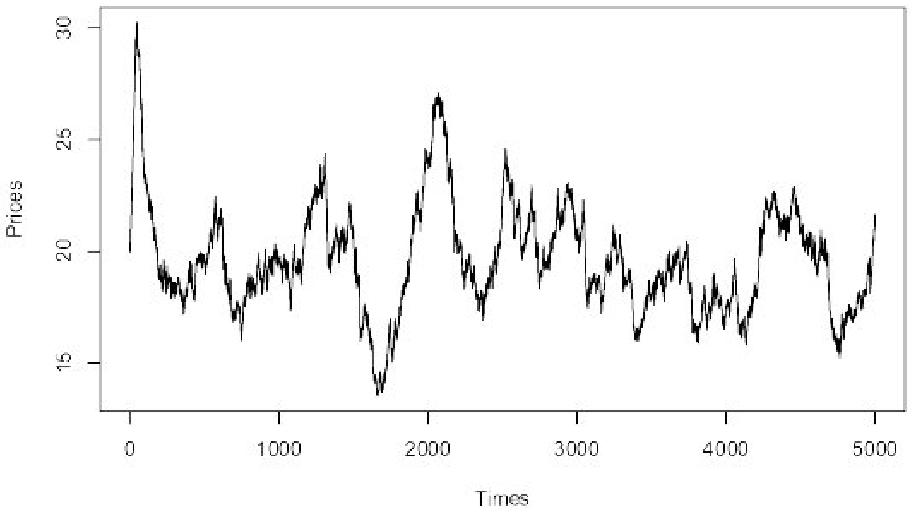


Figure 16: Daily time series for the stock prices

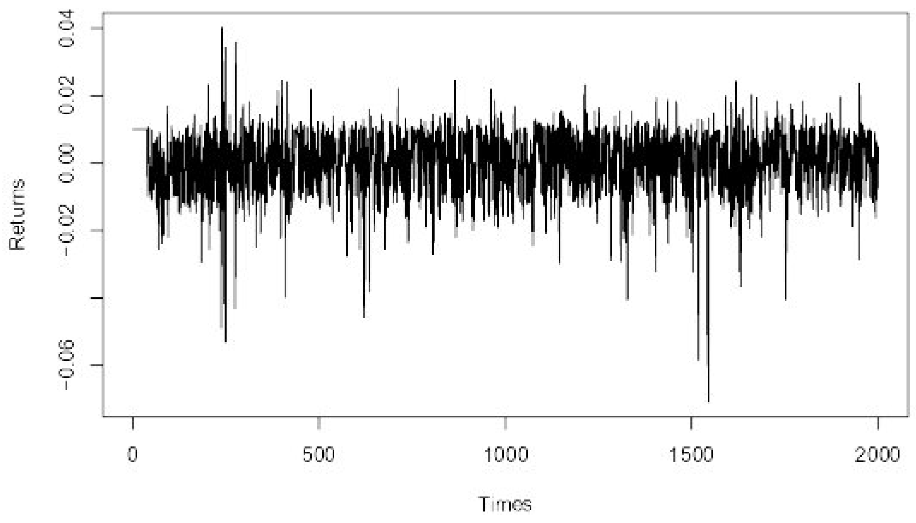


Figure 17: Daily time series for the returns

implementations.

We created a collection of the suites that contains the test-cases used to test the ADT and global functions of the system. These suites are implemented in the following structures, whose names recall those of the tested items: *TestOrder*, *TestPortfolio*, *TestAsset*, *TestDouble*, *TestClearingMechanism*, *TestCollectionOrders, TestTrader, TestClearingHouse*. Note that the tests of the X-Agents (the two latter structures) are inserted into the suite collection, together with other tests.

The ADTs are tested according to what has been reported in section 6.4.2, while X-Agents are tested according to the criteria of section 6.5, and their testing is described in the next section. Fig. 18 shows the test suites written for the system, and their hierarchical organization.

6.5.4 Testing X-Agents

This section reports in detail the test steps followed to build a test-suite for the X-Agents of our simple system.

To test X-Agents, we created a test module for each kind of agent, called: *TestTrader* and *TestClearingHouse*. These modules contain the suites `tests\_Trader` and `tests\_ClearingHouse`, which aggregate the tests cases for every function present in the modules *Trader* and *ClearingHouse*.

In order to perform a test on a particular kind of X-Agent, it is necessary

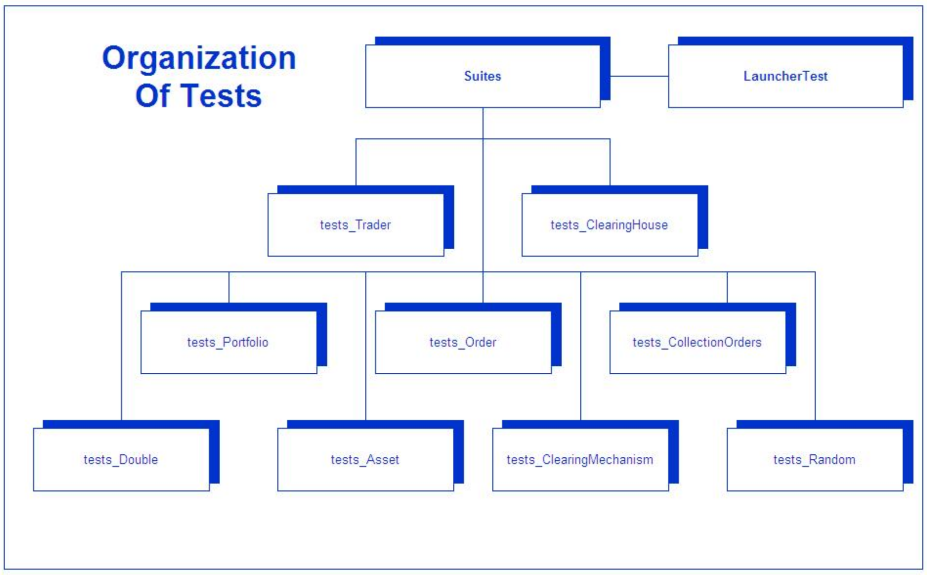


Figure 18: The suites organization

to instantiate a node and to populate the environment with at least one agent of that type. An agent population might be actually dislocated in different nodes, composing a cluster. The agent location should not affect its behavior and its interactions. Conseqently, we consider only a single agent and a single node. The tests of the mechanism of agent migration between different nodes, and of the communication between agents in a distributed system architecture is delegated to the test and validation of the entire Flame environment.

The agent population is created by using the set of functions that is generated automatically by the XMML compiler, in particular the functions managing the instantiation of nodes or of sets of agents.

The function *add_node* allows to instantiate a node, while the functions *add_ClearingHouse_agent* and *add_Trader_agent* are able to instantiate agents of *ClearingHouse* and *Trader* kinds, respectively, adding the new instances into the respective lists.

The creation of the agent pool can be implemented by a particular function, for instance a function `initializePopulation`. This operation should be inserted inside the initialization function of the suite.

The function `initializeEnvironment` contains the initialization of all environment variables that are defined in the file XMML. The function `suite\_Trader\_init` function must call the function that initializes the environment variables, and the function that initializes the message boards (`initialize\_pointers`), as shown in the code that follows:

```
/* Test Case creation for X-Agent Trader
 -- EURACE Project*/

/* Initialization of the test suite for Trader X-Agent */

 int suite_Trader_init(void)
    {
     initialise_pointers();
     add_node(0, -SPINF, SPINF, -SPINF, SPINF, -SPINF, SPINF);
     current_node = *p_node_info;
     initializeEnvironment();
     p_order_message = &current_node->order_messages;
     p_orderStatus_message = &current_node->orderStatus_messages;
     p_xmachine = &current_node->agents;
     initializePopulation();
     return 0;
    }

/* Creation and initialization of global data needed to
   a Trader and a ClearingHouse to work */

 void initializeEnvironment()
   {
    asset = newAsset();
    rnd=newRandom();
    incid = 0;
    pricemechanism=newClearingMechanism();
   }

/* Creation and initialization of a fixture including
   a single Trader with known values for testing */

 void initializePopulation(void)
    {
     int id;
     int tau;
     double wealth;
     Portfolio * portfolio;
     Order * order;
     double posx;
     double posy;
     id = 0;
     tau = 3;
     wealth = 0.0;
     portfolio = newPortfolio();
     order = newOrder(0,0);
     posx = 0.0;
     posy = 0.0;
     add_Trader_agent(id, tau, wealth, portfolio, order, posx, posy);
}
```

Now we have to implement the tests-cases for each function that characterize the behavior of the agent, and for the other support functions. A particular module shall contain the test suite of an X-Agent; for instance, *TestTrader* for the *Trader* module. The transition functions usually modify the state of an agent, so in order to define the fixture, it is necessary to set the agent's state with a given value. Usually, There are three kinds of function to test:

1. functions that have a message as a precondition;
2. functions that send one or more messages;
3. functions that have no precondition and send no message (they modify only the memory of an agent).

We recall that the functions that define the behaviour of a Trader, and are able to send/read messages and to change its state are the following:

1. *selectStrategy* : the trader chooses whether to trade or not, and in the former case choose which kind of order to place (buy or sell), based on some agent's parameters. *selectStrategy* hasn't got any message precondition. In order to test it, it is sufficient to set as fixture a predefined state of the agent and some parameters with known values. This function can modify the agent's state, in the case the trader places an order, transitioning to the *wait* state.
2. *updateTrader:* The trader receives from the clearing house the information on order execution. If the limit price matches the market price, then the transaction is performed, and the trader updates her portfolio. The trader state is changed from *wait* to *inactive.*

The utility functions are following:

- *sendOrder*: Executing this function, the agent decides the limit price and the quantity of stocks to sell or to buy, and sends the order to the clearing house. This function is called from within *selectStrategy* function.
- *estimatedStd*: this function estimates the standard deviation of the logarithms of returns within agiven time window *tau*, which is specific of every Trader. It is called from the previous function and is used to determine the limit price.

- *setBuyLimitedOrder*: set the quantity and the limit price of the buy order.
- *setSellLimitedOrder*:set the quantity and the limit price of the sell order.

As described section 6.5.2, the communication between Trader and ClearingHouse happens through the posting of two types of messages:

1. *order*: it represents the order posted by a trader and sent to the clearing house. It holds the asset quantity, the type (buy or sell), and the limit price.
2. *orderStatus*: this message is posted by the clearing house and sent to a trader using broadcasting. It contains the market price of the asset and the quantity of assets that the order will deal. This quantity is zero if the limit price does not match the cleared price, and is equal to the order quantity in the opposite case.

To perform the tests, it is necessary to create one or more fixtures. In these fixtures we set the agent's state and the prices series of the asset with known values. Since many functions use the environment random variable, we must set the seed, as shown in the following code:

```
/* Service fixture creation for Trader and ClearingHouse
 -- EURACE Project*/

/* Create a given sequence of past prices of the asset */

void setFixForSTD(void) {
    addPrice(asset,20);
    addPrice(asset,21);
    addPrice(asset,22);
    addPrice(asset,19);
    set_tau(2);}

/* Create a given portfolio for a Trader, and set the seed
  of the random number generator */

void setFixureForTesting(void)
 {  Portfolio *port;
    emptyFixedCollection(asset->prices);
    addPrice(asset,50);
    port=get_porfolio();
    setCash(port,5000);
    setQuantity(port,1000);
    set_tau(2);
    setSeed(rnd,2345678901.0);
}
```

In the end, the test suite of the trader is composed by the following test-cases:

***testEstimatedStd*:** tests the function *estimatedStd*. It sets the fixture *setFixForSTD* and verifies the expected value by using assertions.

***testSetBuyLimitedOrder*:** tests the function *setBuyLimitedOrder*. It sets one or more fixtures with a buy order (this is accomplished by calling the *setFixureForTesting* function), and verifies that the order holds the expected values. The fixture sets the agent's state, so an assertion verifies if the state has the expected value.

***testSetSellLimitedOrder*:** tests the function *setSellLimitedOrder*, in the same way as above.

***testSelectStrategy*:** tests the function *selectStrategy*. it verifies that the agents' strategy works correctly, testing that the order holds the expected values. Since strategies are generated at random, it is necessary to set the seed of the random number generator, to get repeatable results.

***testSendOrder*:** tests the function *sendOrder*. It places an order into the communication channel (the message list), and tests that the order is correct.

***testUpdateTrader*:** tests the function *updateTrader*. It sets the agent's portfolio to a given value, sets the agent's state to *wait*, and places an *orderStatus* message into the communication channel, testing that the *updateTrader* function reads correctly the message, and sets accordingly the trader's portfolio. It sets one o more fixtures, and verifies using assertions that the portfolio values are correct.

References

[AK04] ST. Anil Kumar, Jerry Clair. Cunit. *Online: http://cunit.sourceforge.net/*, 2004.

[BB01] J. Odell B. Bauer, J.P. Mller. Agent uml: A formalism for specifying multiagent software systems. *International Journal of Software Engineering and Knowledge Engineering*, vol.11:pp.1–24, 2001.

[BB05] J. Odell B. Bauer. Uml 2.0 and agents: How to build agent-based systems with the new uml standard. *Engineering Applications of Artificial Intelligence*, vol.18:pp.141–157, 2005.

[Bec94] K. Beck. Simple smalltalk testing: With patterns. smalltalk report. *Online: www.xprogramming.com/testfram.htm*, 1994.

[Bec00] K. Beck. Extreme programming explained. 2000.

[Bec01] K. et al. Beck. Manifesto for agile software development. 2001.

[Coh03] M. Cohn. The scrum development process. 2003.

[Cun05] W. Cunningham. Fit: Framework for integrated test. *Online: http://fit.c2.com/*, 2005.

[MSA03] G. Meszaros, S. M. Smith, and J. Andrea. *The Test Automation Manifesto*, volume 2753 of *Lecture Notes in Computer Science*, pages 73–81. Springer Berlin / Heidelberg, September 2003.

[PM03] Poppendieck T. Poppendieck M. Lean softwaredevelopment. 2003.

[PSK88] Miller K. W. Park S. K. Random number generators: Good ones are hard to find. *Communications ACM*, 31(10):1192–1201, 1988.

[Sut03] Jeff Sutherland. Agile project management with scrum: Theory and practice. 2003.