Project no.
035086
Project acronym
EURACE
Project title
An Agent-Based software platform for European economic policy design with heterogeneous interacting agents: new insights from a bottom up approach to economic modelling and simulation

Instrument: STREP

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

Deliverable reference number and title
D6.3: Software module of agent-based models of financial markets
Due date of deliverable:
31/08/2008
Actual submission date:
15/09/2008

Start date of project: September 1^{st} 2006 Duration: 36 months
Organisation name of lead contractor for this deliverable

University of Cagliari-UNICA

Revision 1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	**X**
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

Abstract

This report, which is a deliverable of workpackage WP6, describes the software module implementing an agent-base financial market model, and presents its status at the end of the WP6. The fundamental objective of the WP6 is to design and to develop an artificial financial market using the FLAME framework. This document aims at giving the reader a detailed description of the implementation of the artificial financial market, in the context of the EURACE project and including households, clearing house, firms, bank, government, securities, transactions and relative interactions.
Also included are sections on general model implementation, the guidelines which we followed in order to execute the CUnit testing process, the verification and the validation of the artificial financial market model.

1 Introduction

The main goal of the EURACE project is to model and simulate an agent based dynamic macro-economic system where different markets models will be integrated. Each model is a complex system which can be analyzed and simulated as an evolving system of autonomous interacting agents.

Concerning the AFM (Artificial Financial Market), a first analysis of the model has led to the realization of a modular software structure based on the *"information hiding"* paradigm, both concerning active agents and "utility objects".

Each agent who has an active behavior has been identified as an *X-agent*, that is a model of agent based on stream *X-machine*, a general computational paradigm. On the contraty, "utility objects", which offer services, have been identified as ADTs (Abstract Data Types).

We used an OOA (Object Oriented Analysis) approach, which can be extended to development environments that do not directly support OOP (Object Oriented Programming), because it is essentially based on a separation of modules [1], [7].

We followed the following steps to analyze the AFM module:

- identify the agents and ADTs from the requirements;
- identify the responsibilities of each object;
- identify the collaborations between objects;
- identify the inheritance relationships;
- decompose the system into autonomous subsystem;
- specify the interface of each object;

For each *X-agent*, we have identified the memory variables, a set of transition functions, a set of states and the messages to get the input data, and update its memory and its state.

We recall that the data structure of an *X-agent* can be composed of ADTs, whereas an object can be composed of other objects.
An abstract data type is a data structure provided of a set of operations enabled to work on it. The ADT concept allow to abstract from the implementation details of a data object [6].

In our perspective, ADTs are important because they allow to overcome the inherent limitations of C language, encapsulating complex behavior inside them. Clearly, using ADT amounts to impose a strict discipline on programmers, who are allowed to operate on ADTs only through proper functions, because the C language does not have specific constructs constraining their use, such as the class construct in C++ and Java.

2 Artificial Financial Market Implementation

In this document we provide details of the implementation with FLAME framework, of the Artificial Financial Market model which has been proposed by the Genoa unit.

The AFM model is described in detail in deliverable D6.1 [10], FLAME specifications are described in the deliverable D1.1 [12].

The financial market module should be integrated with various models of the goods, labour, credit and financial management markets, in order to obtain a completely integrated

model as expected in the context of the EURACE project. This is the subject of present integration work.

2.1 Agents Specification

A first bottom-up analysis allowed us to indentify some key agents having active behavior in the financial market model: *Household, ClearingHouse, Firm, Bank, Government.*

For each agent, we also described the memory variables, the messages, the transition functions and the transition states. Furthermore, we identified some objects which have to be implemented as abstract data types.

The agents and the ADTs involved in the implementation are listed and specified in the following sections.

2.2 Household Agent

The financial market we have implemented is a system where the households sell and buy financial securities. Two kinds of financial securities have been considered: *Stock* and *Bond.*

The households trade financial securities in order to obtain profit; their role is to trade assets in exchange for cash, and vice-versa. They issue buy/sell limit orders on the asset (Stock or Bond).

The household owns a portfolio holding all his/her assets. The budget is characterized by an amount of shares and by an amount of cash. The shares amount, owned by each trader, is managed by the *assets_owned* variable; the cash amount is managed by the *payment_account* variable. These are memory variables of the *Household* agent.

On the basis of information on the market behavior, like financial status of firms and their time series stock prices, the households form their own beliefs; namely, they estimate the future asset price at the time step corresponding to their specific *forward window.*
They can follow different behaviors, and namely they can behave as: *random*, *fundamentalist*, *chartist.* Traders belong to four different groups, described in the followings:

Random traders (R): R traders have zero intelligence and simply issue random orders. They are characterized by the simplest trading strategy and represent the "bulk" of traders who do not try to beat the market, but trade for exogenous reasons linked to their needs. If a random trader decides to issue an order, this may be a buy or sell limit order with equal probability.

Fundamentalist traders (F): F traders believe that each asset has got a fundamental price P_f related to factors external to the market and, sooner or later, the price will revert to that fundamental value.

Chartist traders (C): C traders are trend-followers. They are divided into traders who follow the market trend and traders who follow the opposite of the market trend. First play the market following past price trends and strictly rely on price information. They buy (sell) when the price goes up (down). Second speculate, if the stock price is rising, it will stop rising soon and fall, so it is better to sell near the maximum, and vice versa.

The households may decide, according to their behavior, to trade a type of stock instead of another type of stock in order to maximize their profit, or to trade bonds instead of stocks.

At first, the households build their beliefs, that determine "preferences" for each asset. In other words, the household's psicological attitude is modelled by means of the prospect theory [10] (e.g. myopic loss aversion) which determines the "*preferences*" structure:

- Each agent builds for each asset an histogram of past returns in order to evaluate them.
- Then he/she weights the histogram with a prospect value function, that incorporates "myopic loss aversion".
- Prospect utility is mapped into weights for issuing orders.
- The histogram is shifted to take dividend yields into account.
- The histogram is iterated, according to the household's holding period.
- The weighted sum of past returns gives the prospect utility.

Through the "*preferences*" structure, the household computes her/his own desired portfolio and then sends buy/sell orders, for each asset, in order to match the desired portfolio.

Each order is characterized by a *limit price* and a *quantity*. The *limit price* is computed using the last market price and the household's belief; the *quantity* depends on household's "preferences".

The issued orders are added to a *pending orders* collection. The *pending orders* are sent to the *clearing house*, the meeting point for demand and supply, using the *order* message. After that, the household receives a response message, called *order_status*, containing information about the order execution.

Finally, the household updates her/his portfolio and orders, which remain pending until fulfillment or until formation of new beliefs.

2.2.1 Memory

id is a *integer* variable which holds the "identity" of the *Household* agent.

payment_account is a *double* variable which holds the current amount of cash owned by the *Household* agent.

wealth is a *double* variable which returns the total wealth owned by the household; it is the sum of the payment account and the value of each asset owned by the *Household*.

beliefs is a collection[1] of *Belief* ADT which represent the beliefs for each assets.

pendingOrders is a collection of *Order* ADT, which represents the pending orders. Pending orders are orders issued by a trader, but not yet executed, or partially executed.

assetsOwned is an *Asset_array* variable used for managing the assets owned by the Household agent.

assetWeights is a *double_array* variable which holds the weights for each asset owned by the Household.

[1] A collection is a dynamic Array

assetUtilities is a *double_array* variable which holds the utility factors for each asset.

agent's parameters:

forwardWindow is an *integer* variable which represents a forward-looking time-horizon of each household, that indicates the households forward time perspective;

backwardWindow is an *integer* variable that represents a backward-looking time window through which the household looks at the past;

bins is an *integer* variable that returns the number of classes composing the household's hystogram, where past returns of an asset are ordered and grouped;

randomWeigth is a *double* variable which indicates the weight that identifies the random component;

fundamentalWeigth is a *double* variable which indicates the weight that identifies the fundamentalist component;

chartistWeigth is a *double* variable which indicates the weight that identifies the chartist component;

holdingPeriodToForwardW is an *integer* variable that indicates how long an asset is kept by the household;

lossAversion is a *double* variable which sets a particular household's psycological characteristic, modelled by means of *prospect theory* (e.g. *"myopic loss aversion"*). On the base of the *prospect theory* the *preferences* structure is determinated according to the steps:

- Compute the desired portfolio.
- Send buy/sell orders in order to match the desired portfolio.

2.2.2 Functions

The main goal of each household is to build and maintain the expected portfolio, following his/her beliefs.

When an household enters the market, s/he will obtain a portfolio dependent on the market state; so, the expected portfolio is only partially matched to her/his belief.

In order to achieve the desidered portfolio, each Household agent executes the following actions:

Household_receive_info_interest_from_bank the *Household* agent receives account interest information from the *Bank* agent.

Household_select_strategy at each time step, which roughly corresponds to a day of trading, each trader decides whether to update or not her/his beliefs, on the basis of the information received by the firms. In the first case, the household computes the new expected portfolio and creates the new pending orders with a given probability set to a given value p_co (presently, this value is 10%), or the old pending orders are kept with a probability set to $1 - p_co$.
In order to execute this transition function, the following utility functions have to be activated:

- **generatePendingOrders** new pending orders are created;

- **assets_beliefs_formation** the belief formation mechanism for each asset is implemented;
- **computeUtilities** asset utilities are computed;
- **assetUtilitiesToWeights** weights formation is made, accordingly to household's choices related to the set of orders;
- **sendOrders** the Household decides whether to update or not his/her beliefs. In both cases, invoking this utility function, s/he sends the pending orders to the ClearingHouse agent via *messages*. Afterward, the household receives a response message containing information about the order execution. An order can be executed totally or partially, or can remain un-executed.

Household_update_its_portfolio the Household agent updates his/her portfolio and orders, which remain pending until full fulfillment, or new belief formation. The portfolio updating happens after receiving the command to execute orders. If a transaction happened, then the Portfolio has to be updated; in other words, the stock quantity and the bank account have to be updated. In this phase of the simulation, household can get or not his/her expected Portfolio.

2.2.3 Messages

accountInterest the household receives from the Bank a response message called *accountInterest*, containing information about his/her account interest. This is an input message to the transition function *Household_receive_info_interest_from_bank*;

info_firm household agents receive information about firm's financial status and stock prices time series using *info_firm* message. This is an input message to the transition function *Household_select_strategy*;

info_bond household agents receive information from the *Government* agent about the bonds, using *info_firm* message. This is an input message to the transition function *Household_select_strategy*;

order through the *order* message, the orders are added to the pending orders collection that is sent to the *ClearingHouse* agent. This is an output message from the transition function *Household_select_strategy*;

order_status the households receive from the *ClearingHouse* a response message called *order_status*, containing information about the order execution. This is an input message to the transition function *Household_update_its_portfolio*;

bankAccountUpdate through the *bankAccountUpdate* message, the household's bank account is updated. This is an output message from transition function *Household_update_its_portfolio*.

2.2.4 States

START_HOUSEHOLD the household starts his/her trading activity in the financial market. The *Household* receives information about his/her account interest from the Bank

agent, through the input message called *accountInterest*. This message causes the execution of the transition function *Household_receive_info_interest_from_bank* and the household switches from START_HOUSEHOLD state to SELECT_STRATEGY state;

SELECT_STRATEGY firm agents send information about their financial status and their stock prices time series using *info_firm* message. The *info_firm* message causes the execution of the transition function *Household_select_strategy* that generates the output message *order*. The Household switches from SELECT_STRATEGY state to WAIT_ORDER_STATUS state;

WAIT_ORDER_STATUS the *Household* agent receives the *order_status* message from the *ClearingHouse* agent and then updates his/her portfolio through the execution of the transition function *Household_update_its_portfolio*. This function generates the *bankAccountUpdate* output message.
The Household agent switches from WAIT_ORDER_STATUS state
to START_HOUSEHOLD_LABOUR_ROLE state.

START_HOUSEHOLD_LABOUR_ROLE when the *Household* agent is in this state, his/her trading activity in the financial market ends, and s/he switches to the labour market

2.3 ClearingHouse Agent

The *ClearingHouse* is an agent that implements the stock and the bond price formation process. The price formation process is centralized and modelled according the clearing house mechanism.

Buying and selling orders, issued by the households, are collected by the clearing house. These orders are sent to an ADT called *ClearingMechanism* that builds the cumulative demand and supply curve.

The market price of the asset is based on the intersection of the demand and supply curve. The crossing point between demand and supply curve is chosen in order to maximize the transactions' amount. The limit orders are issued on the asset.

The *ClearingHouse* agent selects the limit orders for each traded stock type. For example, if the traders issue limit orders on the "Barclays" stock and on the "UBS" Stock, the clearing house will separate the "Barclays" stock orders from the "UBS" stock orders.

As soon as the new stock price is computed, the exchanges between *sellers* and *buyers* will happen if the sell limit order price matches the buy limit order price. The exchange happens at the cleared price.

2.3.1 Memory

assets *Asset_array* variable, which rapresents the list of assets that are traded in the simulations;

clearingMechanism istance of the *ClearingMechanism* ADT.

2.3.2 Functions

ClearingHouse_receive_info_stock *ClearingHouse* agent gets information from the Firm agent; the *Firm* agent has a stock database from which the *ClearingHouse* agent gets the information about different stocks. The stocks number may change dynamically. In fact, when a firm goes bankrupt, the related stock is deleted from the database; when a firm enters the stock market, the related asset is added to the database. The *ClearingHouse* gets information also about the last price value of the *Firm*.

ClearingHouse_receive_orders_and_run the *ClearingHouse* receives the orders and then discriminates them by type of stock. The computation of asset prices is delegated to the *ClearingMechanism* ADT. The *ClearingHouse_receive_orders_and_run* function is described by the following steps:

- **ReceiveOrdersOnAsset** divides the orders on the base of different Stock types;
- **ComputeAssetPrice**: this function computes the new price value of each asset and executes the *rationing*[2] of the orders;
- **SendOrderStatus**: sends messages about the execution state of the orders to the *Household* agent.

ClearingHouse_send_asset_information creates an *info_Asset_CH* message for each Stock that contains the informations about new Stock price and then sends the *info_asset_CH* messages to Firm agent.

2.3.3 Messages

info_firm the *ClearingHouse* agent receives information about the stock prices time series from Firm agent. The *info_firm* message is an input message of the transition function *ClearingHouse_receive_info_stock*.

order the *ClearingHouse* receives the orders from the households. The *order* message is an input message of the transition function *ClearingHouse_receive_orders_and_run*.

order_status the *ClearingHouse* agent sends the response message to the Household agent containing information about the issued orders execution state.
The *order_status* message is an output message of the transition function *ClearingHouse_receive_orders_and_run*

info_Asset_CH the *ClearingHouse* agent creates the *info_Asset_CH* message, for each asset, containing information about the new asset price; *info_Asset_CH* message is sent from ClearingHouse agent to Firm agent after that the new stock price is been computed, so the Firm agent is updated on last price value. This is an output message from the transition function *ClearingHouse_send_asset_information*.

2.3.4 States

START_CLEARINGHOUSE The *ClearingHouse* agent starts its trading activity in the financial market. The *ClearingHouse* agent receives information about the stocks and

[2]it balances the quantity of demand and the quantity of supply

the bonds rispectly from the *Firm* and from the *Government* agents through the input messages *info_firm* and *info_bond*.
These messages causes the execution of the transition function *ClearingHouse_receive_info* and the switch from the START_CLEARINGHOUSE state to RECEIVED_INFO_STOCK state.

RECEIVED_INFO_STOCK The *ClearingHouse* agent receives the orders from both the *Household* agent and from the *Government* agent through the input message called *order*.
The *order* message activates the transition function *ClearingHouse_receive_orders_and_run*. The orders are collected on the basis of different types of assets, and then the new asset price is computed. This transition function generates the output message *order_status*, which informs the *Household* agent about the order execution state. The *ClearingHouse* agent switches from RECEIVED_INFO_STOCK state to COMPUTED_PRICE state.

COMPUTED_PRICES After computing the new asset price, the transition function *ClearingHouse_send_asset_information* is activated. The *ClearingHouse* agent creates the output message *info_Asset_CH* and sends information about stock and bond to *Government* agent and to *Firm* agents, rispectively.
The agent then switches to the END_CLEARINGHOUSE state.

END_CLEARINGHOUSE the *ClearingHouse* agent ends its activity in the financial market.

2.4 Firm Agent

The *Firm* is an agent which may own and trade stocks for specific operating purposes, like for increasing its liquidity.

2.4.1 Memory

id *integer* that holds the "identity" of the specific *Firm* agent.

earnings *double* variable holding the revenues minus total production costs. The revenue is the income produced by a particular source

earnings_payout *double* variable holding the number of dividends to pay

equity *double* variable holding total assets minus total debt

current_shares_outstanding *double* variable holding the number of shares owned by the Firm

current_dividend_per_share *double* variable holding the total dividends divided by the number of shares

stock istance of the *Stock* ADT, which represents the stock entity of a given *Firm* and carries information about time series of prices and returns

2.4.2 Functions

Firm_send_info the *ClearingHouse* agent gets information about stock prices and the *Household* agent gets information about financial status of the Firm and also the prices of the stock owned by the Firm

Firm_receive_stock_info throught the *info_Asset_CH* message, the Firm agent receives information about the new stock price by the *ClearingHouse* agent.

2.4.3 Messages

info_firm this is an output message from the *Firm_send_info* transition function sent to the *ClearingHouse* agent and to the *Household* agent, containing information about earnings, dividend per share, equity, earnings payout and the specific stock issued on the market. This informations is used by the *Household* agents in order to form his/her beliefs about the fundamental price of each stock.

info_Asset_CH this is an input message sent by the *ClearingHouse* agent to the *Firm* agent, containing information about the new market price of the specific stock issued by the *Firm* agent

2.4.4 States

START_FIRM the *Firm* agent starts its trading activity in the financial market; at the start of each financial market session, the *Firm* sends the *info_firm* message through the transition function *firm_send_info* to *ClearingHouse* and *Household* agents, which receive information about the specific stock and the *Firm* financial status

RECEIVED_STOCK_INFO at the end of each financial market session, the *Firm* receives the *info_Asset_CH* message sent by *ClearingHouse*, then updates the time series of stock prices agent

END_FIRM the *Firm* agent ends its activity in financial market.

2.5 Bank Agent

The *Bank* is an agent that receives funds from *Household* agents, and use them to lend. *Firm* and *Household* agents, when in the need of borrowing money, seek a subset of Banks, on the basis the interest rate.

2.5.1 Memory

id *integer* that holds the "identity" of the *Bank* agent.

2.5.2 Functions

Bank_send_accountInterest the *Household* agent gets information about his/her account interest from the *Bank* agent, using *accountInterest* message

2.5.3 Messages

accountInterest the *Household* agent receives information about the account interest rate from the *Bank* agent. This is an output message from transition function *Bank_send_accountInterest*.

2.5.4 States

START_BANK the *Bank* agent starts its activity in the financial market. The *Bank* sends information about its account interest to the *Household* agent throught the output message called *accountInterest*.
The transition function *Bank_send_accountInterest* is activated and the *Bank* agent switches from START_BANK state to BANK_START_CREDIT_MARKET_ROLE state

BANK_START_CREDIT_MARKET_ROLE the *Bank* agent starts its activity in the credit market

2.6 Government Agent

The *Government* is an agent which may issue both short-term or long-term bonds in order to finance its budget deficit.

2.6.1 Memory

id *integer* variable that identifies the "identity" of the *Government*.

bond istance of the *Bond* ADT.

payment_account *double* variable that holds the total liquidity owned by *Government* owned in the Central Bank.

pending_order is a collection of *Order* ADT, which represents the pending orders on the bonds. Pending orders are orders issued by a trader, but not yet executed, or executed only partially.

deficit *double* variable that returns the deficit of the *Government* agent.

day_of_month_to_act *integer* variable that holds the time step on which the *Government* agent acts.

2.6.2 Functions

Government_send_info_bond creates an output message called *info_bond* that contains the information about the *Bond*. The *Government* agent sends this message to *Household* agent, who will buid its trading strategy and then to *ClearingHouse* agent who gets information about the bonds, in order to compute the new price of the bond.

Government_orders_issuing if the deficit is < 0, the *Government* agent issues orders whith a bond quantity that depends on its deficit and on the bond price. The issued orders are added to a pending orders collection. This transition function sends an output message called *order* to the *ClearingHouse* agent.

Government_update_its_portfolio the *Government* agent updates its portfolio and its orders. The updating of the portfolio happens after receiving the input message *order_status* from the *ClearingHouse* agent.

Government_receive_info_bond the *Government* agent receives information about the new bond price throught the input message *info_Asset_CH*

2.6.3 Messages

info_bond the *ClearingHouse* and the *Household* agents receive information about the Bonds issued by the Governament agent. This is an output message from the *Government_send_info_bond* transition function.

order the *ClearingHouse* receives the orders from the *Government* agent throught the output message called *order* from *Government_orders_issuing*.

order_status the *Government* receives infomation from the *ClearingHouse* agent about the issued orders execution state. The *order_status* is an input message of the transition function *Government_update_its_portfolio*.

info_Asset_CH the *ClearingHouse* agent creates the *info_Asset_CH* message, containing information about the new price of the bond. This is an input message to the *Government_receive_info_bond* transition function

2.6.4 State

START_GOVERNMENT the *Government* agent starts its trading activity in the financial market.

GOVERNMENT_SENT_INFO_BOND the *Government* sends information about the Bond to the *Household* and to *ClearingHouse* agents using the message called *info_bond*. This is the output message from the transition function *Government_send_info_bond*, that causes the switch from START_GOVERNMENT state
to GOVERNMENT_SENT_INFO_BOND state.

GOVERNMENT_SENT_ORDER the *Government_orders_issuing* function is activated; the output mesage *order* is sent to the *ClearingHouse* agent and the *Government* switches to GOVERNMENT_SENT_ORDER state

GOVERNMENT_PORTFOLIO_UPDATED The *Government* receives from the *ClearingHouse* the *order_status* message that contains information about order execution state, and then updates its portfolio throught the activation of the
Government_update_its_portfolio function.
The agent switches to GOVERNMENT_PORTFOLIO_UPDATED state.

END_GOVERNMENT When the agent is in this state, it has ended its activity in the financial market

3 ADT

In the presented model, ADTs are data structures, with a set of related C functions, providing non-trivial services to agents.

An ADT is typically contained in the data structure of an agent, in place of a set of scattered variables logically linked together. For instance, the *Stock* ADT provides all the information on a given stock, including its price series, liquidity, past returns.

The same result could have obtained putting directly all this information in the data structure of the firm issuing the stock, but with much lower modularity and ease of future extensions.

The following ADTs have been identified:

3.1 Asset

We have defined an ADT called *Asset* representing the ownership of a financial products of the AFM model by a *Household*, a *Firm* or other agents.

Each *Asset* knows its money value, its last price value, the quantity owned by the *Household* or issued by the *Firm* on the market.

3.1.1 Variables

id *integer* variable which identifies the "identity" of the firm that issues the specific financial product.

quantity *integer* variable that indicates the quantity of the asset that is issued by the *Firm* on the market or owned by the *Household*.

lastPrice *double* variable that holds the last asset price value.

3.2 Stock

The *Stock* ADT is a specific financial security owned by a specific firm. It represents the kind of the financial product that is issued by the firm.

3.2.1 Variables

id *integer* variable that identifies the *Stock*.

nrOutStandingShares *integer* variable that identifies the number of shares of a specific Stock owned by a firm.

prices[HISTPRICES_LENGHT] *double_array* collecting the historical prices of the *Stock*.

returns[HISTRETURNS_LENGHT] *double_array* collecting the historical returns of the *Stock*.

3.3 Bond

The *Bond* ADT represents a specific debt security issued by the *Government* agent.

3.3.1 Variables

id *integer* variable that identifies the specific *Bond*.

nr_outstanding *integer* variable that holds the number of shares of a specific *Bond* owned by the *Government*.

nr_quantity *integer* variable which holds the *Bond* quantity that may still be traded.

face_value *double* variable which holds the amount of money the *Government* will get back to *Household* once a *Bond* expires.

nominal_yield *double* variable that is the interest rate stated on the face of the *Bond*, which represents the percentage of interest to be paid by the *Government* on the *face_value* of the *Bond*.

maturity_day *integer* variable which represents the date on which the issuer has to repay the nominal amount.

issue_day *integer* variable date on which the bond is issued.

prices[**HISTPRICES_LENGHT**] *double_array* which collects the historical prices at which investors buy the specific *Bond*.

returns[**HISTRETURNS_LENGHT**] *double_array* which collects the historical returns of the specific *Bond*.

3.4 Order

Order is an ADT which represents a buy or sell limit order for a given security.

3.4.1 Variables

issuer *integer* variable holding the "identity" of the *Household* who issued the order itself.

limitPrice *double* variable that rapresents the limit price of the issued order.

quantity *integer* variable that holds the quantity of the issued order.

assetId *integer* variable denoting the identity of the asset on which an order is issued.

3.5 Belief

On the base of financial status of the firms and their asset prices time series, the households form their own beliefs; namely they estimate the future assets price at the their specific "*forward window*".

3.5.1 Variables

asset_id *integer* variable holding the "identity" of the asset (bond or stock).

expectedPriceReturns *double* variable holding the expected value of the price returns computed on a past time window with *backwardWindow* lenght.

expectedTotalReturns *double* variable holding the expected value of the total returns computed on a past time window with *backwardWindow* lenght. The total returns are computed as the sum of price returns and dividend returns (for the stocks), or of coupons (for the bonds).

expectedCashFlowYield *double* variable holding the expected dividends on a future time window. The window lenght is equal to *forwardWindow*.

volatility *double* variable holding the volatility computed using the *backwardWindow*.

expectedEarning *double* variable holding the earnings expected value.

expectedEarningPayout *double* variable holding the expected value of the earnings payout. The earnings payout refers to the portion of net income which is distribuited to owners as dividend.

lastPrice *double* variable holding the last asset price which the istance of *Belief* refers to.

utility *double* variable holding the asset utility.

3.6 ClearingMechanism

The *ClearingMechanism* is an ADT which collects all orders sent by the *ClearingHouse* agent, and then builds the cumulative demand and supply curve.

3.6.1 Variables

lastPrice *double* variable holding the new market price of the asset.

sellOrders *Order_array* variable holding all sell limit orders issued by households.

buyOrders *Order_array* variable holding all buy limit orders issued by households.

prices *double_array* variable holding the historical asset prices.

4 State dependency diagram

The state graph in Fig. 1 shows the flow of activity in the complete AFM model.

The figure depicts for each agent a set of transition functions, and a set of states and messages. Messages get the input data and, depending on them, cause the execution of a specific transition function which updates the memory and the state of agents.

In the diagram, the states are represented as ellipses, the transition functions as rectangles and the messages as arrows. The activities flow occurs through several layers (layer 0, layer 1,...,layer n).

Layer 0 denotes the entry of a specific agent in the AFM and the starting of her/his activity; the final layer denotes the end of activity in the AFM and a possible swicth to another market.
For instance, in layer 4 the *Household* agent is in the START_HOUSEHOLD_LABOUR_ROLE state; s/he leaves the AFM and enters the Labor market, while in layer 5 the *Government* agent ends its activity in the AFM.

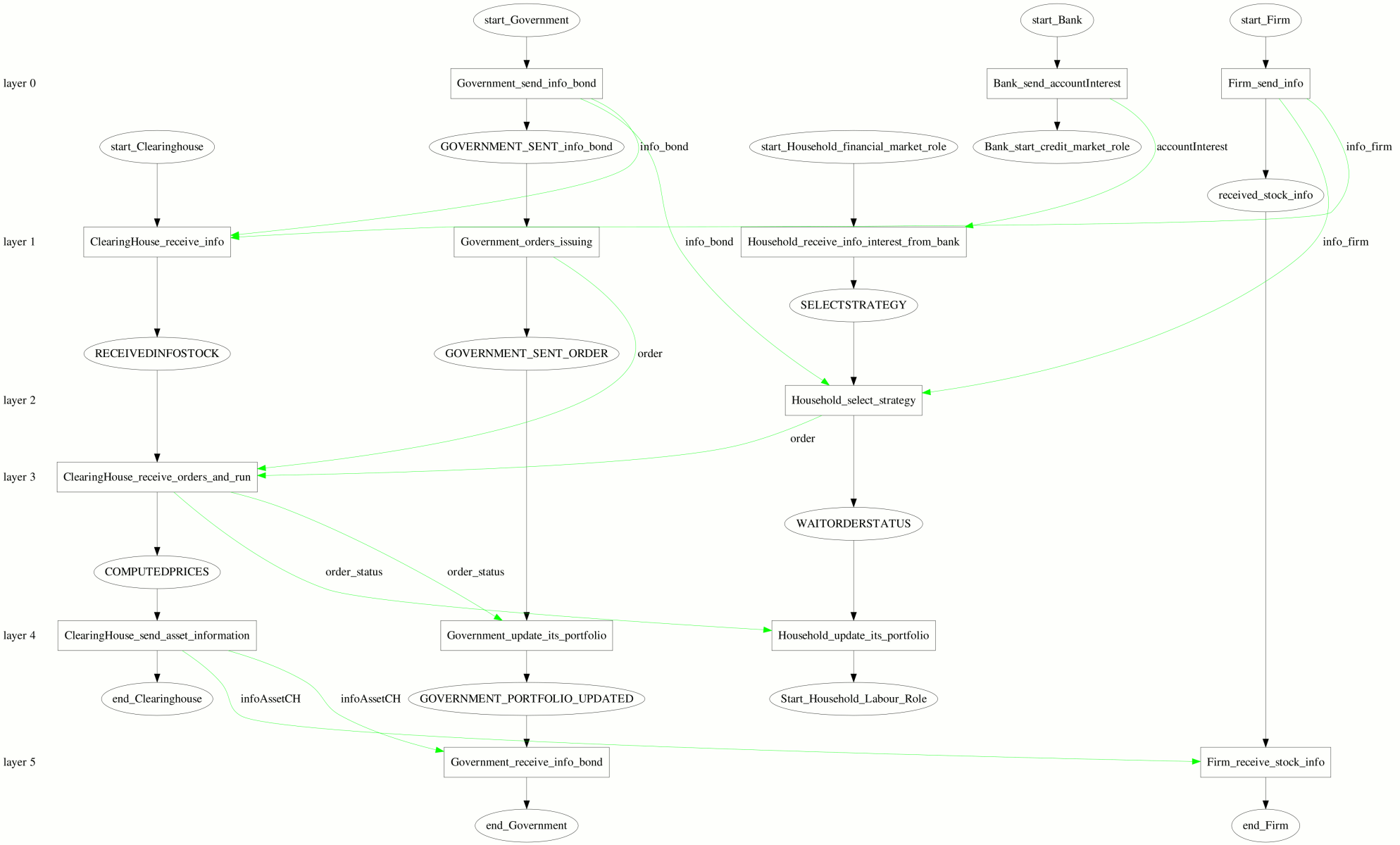

Figure 1. Dependency diagram of the AFM module.

5 Verification and Validation of the AFM model

One of the most critical issues in developing software simulation models is the verification and validation of the model itself.

Model verification usually refers to techniques used to ensure that the computer programming and implementation of the theoretical model are correct.

Concerning the verification of the AFM model, we used a process testing based on the CUnit in order to test the transition functions used in the AFM model just now. Obviously the testing process will have to be extended to each function or procedure.

Model validation consists in understanding "*if the computerized model within its domain of applicability possesses a satisfactory range of accuracy consistent with the intended application of the model*"[9].

Artificial financial markets can be characterized by a large number of parameters in order to fit any kind of real data, but this approach usually leads to complicated models or at least models extremely dificult to calibrate. A rule of thumb is to keep the model as simple as possible, leaving out all unnecessary components.

The problem of validation can be addressed with the requirement that the implemented model exhibits the main statistical properties of financial time series called *"stylized facts"*. The price time series produced by the artificial markets have to exhibit the same statistical features of real markets. This is not a suggestion, it is the necessary condition that allows to validate the model.

In the following subsections we will briefly list the most common *"stylized facts"*, the verification of the model using CUnit testing process and the validation of the AFM implemented with FLAME framework.

5.1 "Stylized Facts"

It is well known that the economic time series of almost all financial assets exhibit a number of non trivial statistical properties called *"stylized facts"*. For a complete discussion about *"stylized facts"* and statistical issues see Pagan [8], Cont [3], Farmer [5], Bouchaud [2].

There is a set of *"stylized facts"* which appear to be the most important and common to a wide set of financial assets:

- *Random walk property of prices*: financial asset prices evolve according to a random walk and thus the prices of the asset market cannot be predicted;
- *Fat tails of returns*: empirical studies generally concur that at weekly, daily and higher frequencies return distributions consistently exhibit more probability mass in the tails and in the centre of the distribution than does the standard Normal. So, the most important finding is that the distribution of returns is non Gaussian and heavy tailed;
- *Volatility clustering*: volatility measures the amplitude of price fluctuation of a financial instrument within a specific time horizon. Volatility is often estimated by calculating the standard deviation of the price values in a certain time window. In the time series of real stock prices, it is observed that the variance of returns or log-prices is high for extended periods and then low for subsequent extended periods: this phenomenon is called volatility clustering. Volatility clustering is strictly correlated with two more dependence properties of returns financial time series: the absence of linear autocorrelation and the presence of non linear autocorrelation. In the first case the autocorrelation

of raw returns is often insignificant, except for very small intraday time scales; in the second case the autocorrelation of absolute returns and of their square, display a positive and slowly decaying autocorrelation, ranging from a few minutes to a several weeks [4]. This phenomenon can be considered as a quantitative manifestation of the volatility clustering itself, and suggests that bursts of volatility can persist for periods that range from hours to days, weeks or even months.

5.2 CUnit testing process: Verification

Following the assumptions made in deliverable D1.2 [11], in order to test the AFM model implemented with FLAME, we use CUnit, a testing framework developed for C code modules.

CUnit offers a set of headers and libraries that helps to automatically execute user-defined tests, making easier to keep under control the system evolution.

In order to write CUnit tests, there are some steps to follow:

1. First of all, it is necessary to initialize the testing "environment"; this can be done using a main function, invoking the functions that create the test registry and a test suite that contains all the unit tests; this suite can be executed every time the user needs to run his tests.

   It is also important to initialize the FLAME enviroment, such as global variables and message boards, invoking those functions already present in the system.

   This step initializes an instance of the AFM to a known state, so that it is possible to run tests, obtaining pre-defined results if the tests are correctly passed.

2. The other item needed to run tests, is the test code itself; it is simply made by functions, that must follow some naming conventions, that are automatically invoked by the test suite.
   A typical test function is divided in 3 major sections:

   - initialization of the local variables and test pre-conditions;
   - execution of the function to test, and check of the post-conditions;
   - clean-up of the memory, in order to run new tests (i.e. memory deallocation, variables reset, and so on...).

All the EURACE system is agent-based, so the first thing to do is to create one or more new agents (only those needed in our test), initialize them and then add these new agents to the system.These agents should also be referred to by pointers that trace these agents acting in the environment.

The agents have their own variables, but only those needed in the test context should be initialized. The same consideration is valid also for global variables.

Then, if our test needs to interact with the Message Board (MB), it is important to create the MB and to check its correctness. If needed, it is possible to add pre-defined messages to the message board, so agents can use them in their functions. This is important because in this way it is possible to test the interaction between other EURACE modules, that communicate each others using messages.

The next step involves the evaluation of the desired function and the "*assertions*" checking whether the function works properly. This is done directly invoking the function and then

using the assertion methods already existing in the framework, that allow users to compare variable values and pre-set values fixed by programmers.

The test code is compiled together with the C files that contain the model to test, the CUnit library and the messageboard library.

The test executable is then run; it outputs a brief summary of the results, such as the test executed and the number of test failed and passed.
Figure 1 shows an example of unit test process executed on the transition function called *Household_select_strategy*, used in the AFM model.

The *Firm* agent sends information about its financial status and its stock prices time series using *info_firm* message to the *Household* agent.

The *info_firm* message causes the execution of the *Household_select_strategy* transition function that generates the output message called *order*. Through this message, *order* is added to the pending orders collection that is sent to the *ClearingHouse*. The *Household* agent switches from SELECT_STRATEGY state to WAIT_ORDER_STATUS state.

On the base of the preconditions and the input messages, we verified that the transition function considered generates the correct answer from our test data. In fact the *order* output message composed by the limit price and quantity to trade, is built and is inserted into the Message Board.

TRANSITION FUNCTION TEST STRUCTURE

PRECONDITIONS	CURRENT STATE	INPUT MESS.	TRAN.FUNC	NEXT STATE	POSTCONDITIONS	OUTPUT MESS.
payment_Account=5000	SELECT_STRATEGY	info_firm (*)	Household_ select_ strategy	WAIT_ ORDER_ STATUS	payment_Account=5000	order
forwardWindow=20		prices[100]={random distr.}[1]			forwardWindow=20	limit_price=52
backwardWindow=5		returns[100]={random distr.}[2]			backwardWindow=5	quantity=94
randomWeight=1		nrOutStandingShares= 300			randomWeight=1	
fundamentalWeight=0					fundamentalWeight=0	
chartistWeight=0					chartistWeight=0	
bins=1					bins=1	
lossAversion=2.25					lossAversion=2.25	

(*) *info_firm* message is composed by the variable 1 and 2. The random values assigned to the static arrays influences the limit price and quantity of the order; in this case, prices are distributed around 50 and returns have 0-mean

Figure 1: Test structure of the *Household_select_strategy* transition function

5.3 Experimental results: Validation

We present some simulations that have been performed with the FLAME implementation of the AFM model.

In order to investigate the validity of the implementation, we ran an isolated model of a closed market and we analyzed the time series of the stocks prices and bond prices.

Our goal is to validate the model implemented with FLAME framework. We ran the simulation process in order to understand if the model is capable to reproduce the main

"stylized facts" of the financial markets.

We considered an economy with two type of assets: bonds and stocks.

In all cases, the Government pays coupons and the Firms pay dividends, but we assume that the earned cash is spent elsewhere, so that the economy is closed: the total amount of cash C and stock S available in the economy remains constant over time.

Each simulation is performed with $N = 2000$ time steps and $M = 1000$ traders.

We performed many test runs varying the percentage of trading population. In a first round of simulations, the agent population is composed of 90% random traders and the remaining 10% is equally divided between chartists and fundamentalists.

We considered $K = 2$ different stock markets, one for each *Firm*, and only one bond.

Each trader i is initially endowed with a quantity c_i of cash, a quantity $s_{i,k}$ of stock k and a quantity b_i of bond.

Stocks dividends, bond coupons and interest on the bank accounts are distribuited every six mounths, but this distribuition influences only the beliefs formation, beacause we supposed that there isn't any exogenus flow of cash.

The bond pays a semestrial coupon of 2% with respect to its face_value.
The dividends evolve according to an exogenous stochastic process, estimated by the equation:1

$$\log d_{\tau}^{a} = \log d_{\tau-1}^{a} + g^{a} + \xi_{\tau}^{a} * \sigma^{a} \tag{1}$$

where g_a is the characteristic growth of the dividend of asset a, σ^a is its characteristic standard deviation, and ξ_{τ}^{a} is a gaussian noise term with zero mean and unitary variance affecting the process at day τ, (see deliverable D6.1 [10]). Furthermore, the Households do not receive any compensation for their work.

We initialized other variable such as:

$$\begin{aligned} payment\_account &= 50000; \\ asset\ amount &= 1000; \\ Bank\ rate &= 0.01 \\ forwardWindow &= uniform(10, 60) \\ backwordWindow &= uniform(10, 20) \\ earnings\_payout &= 10\%\ of\ earnings \\ &... \end{aligned}$$

Figure 2 shows the daily time series prices of the Stock with $id = 0$, Figure 3 and 4 show the autocorrelations of returns and absolute returns, respectively.

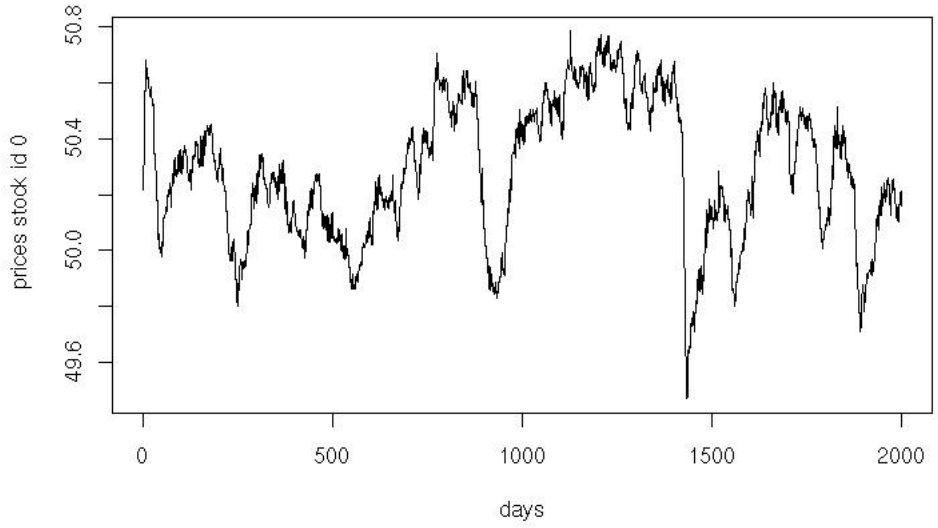

Figure 2: Time series of prices of Stock 0

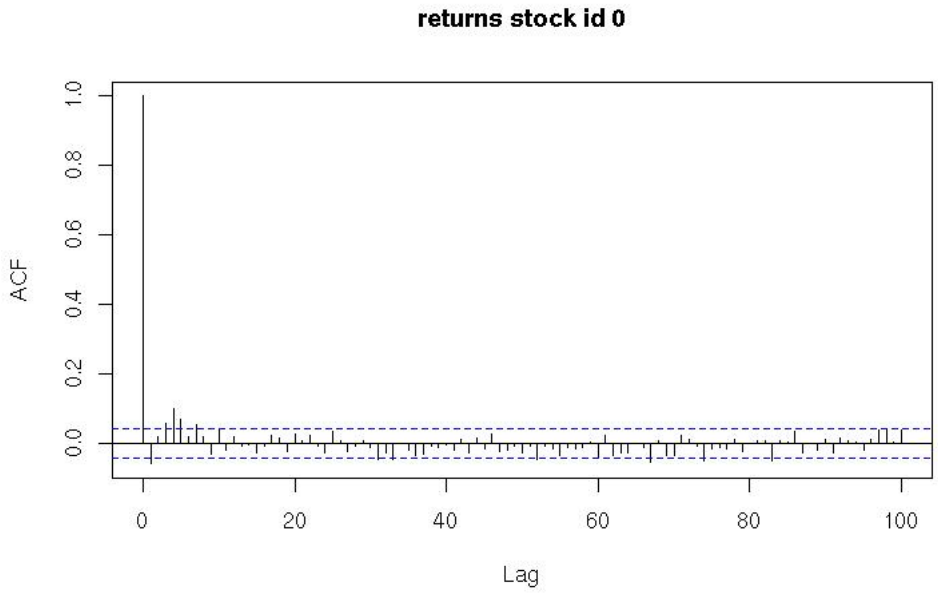

Figure 3: Autocorrelation of returns of Stock 0

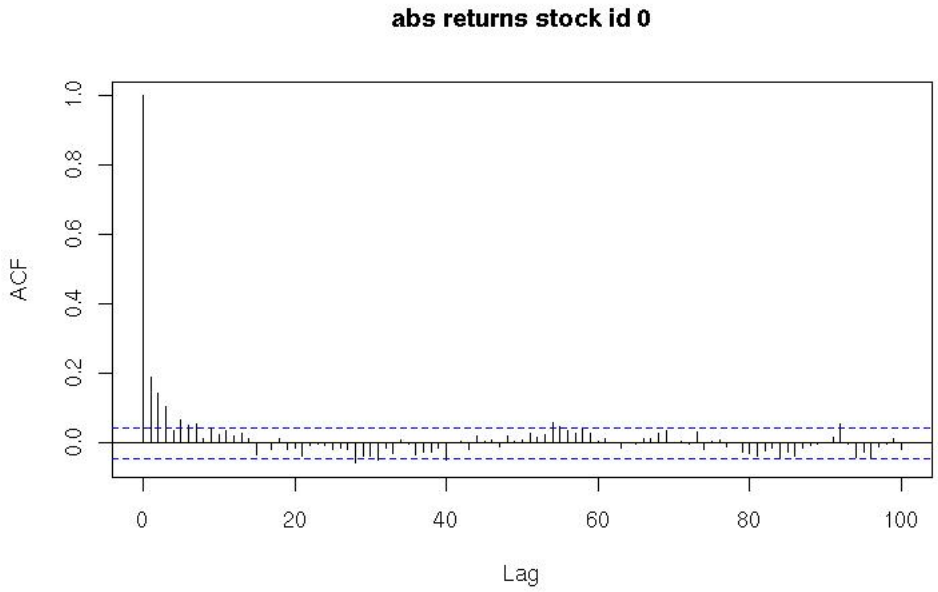

Figure 4: Autocorrelation of absolute returns of Stock 0

Figure 5 shows the daily time series prices of the Stock with $id = 1$, Figure 6 and 7 show the autocorrelations of returns and absolute returns, respectively.

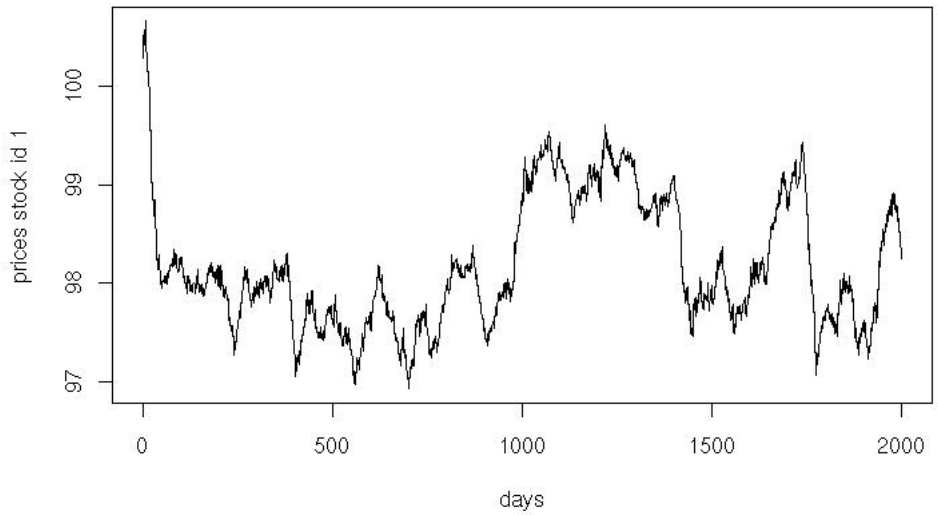

Figure 5: Time series of prices of Stock 1

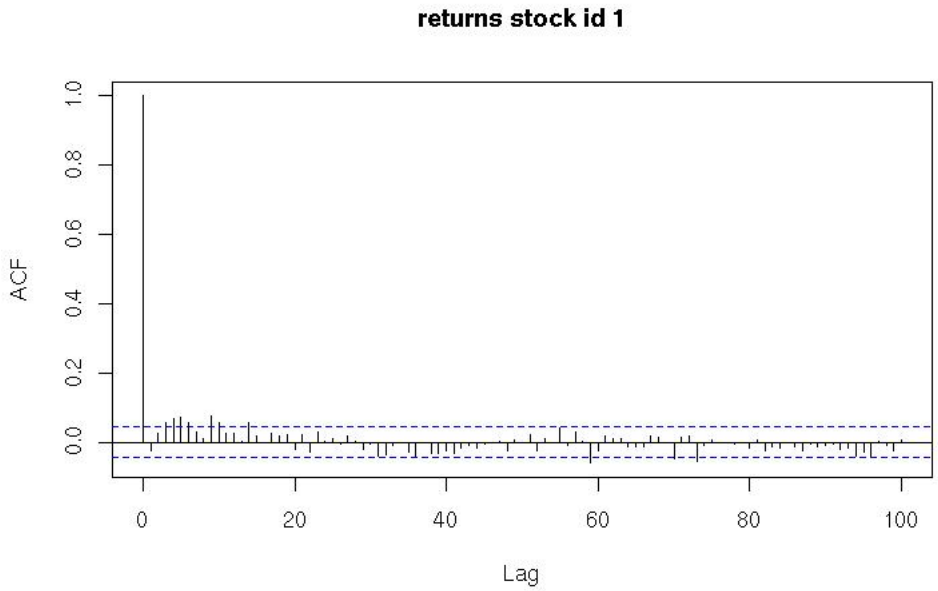

Figure 6: Autocorrelation of returns of Stock 1

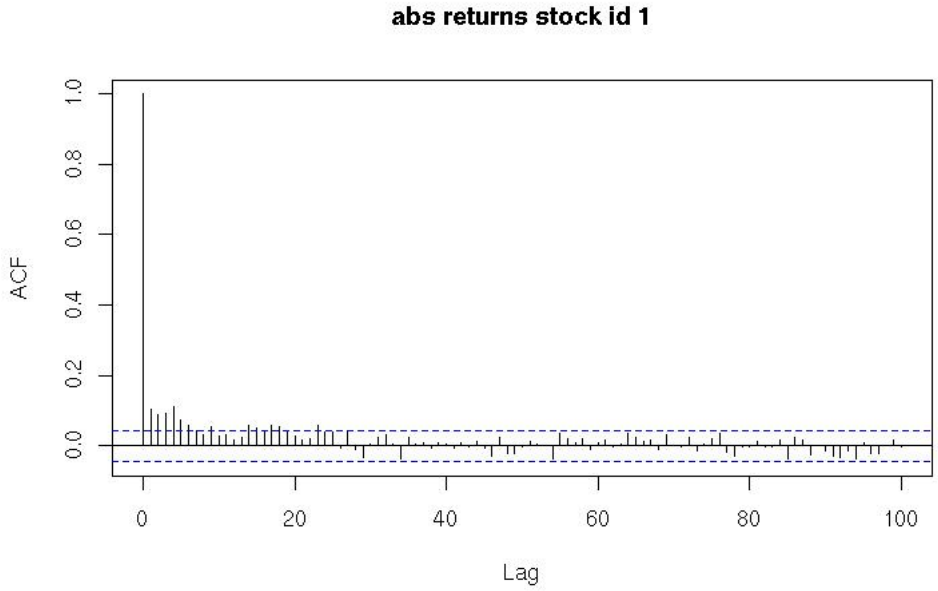

Figure 7: Autocorrelation of absolute returns of Stock 1

Figure 8 shows the daily time series prices of the Bond, Figure 9 and 10 show the autocorrelations of returns and absolute returns, respectively.

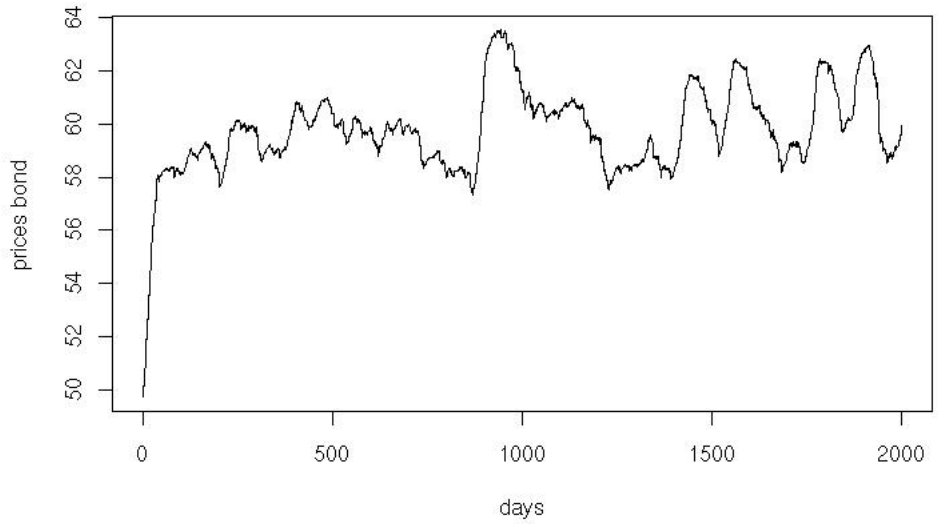

Figure 8: Time series of prices of the Bond

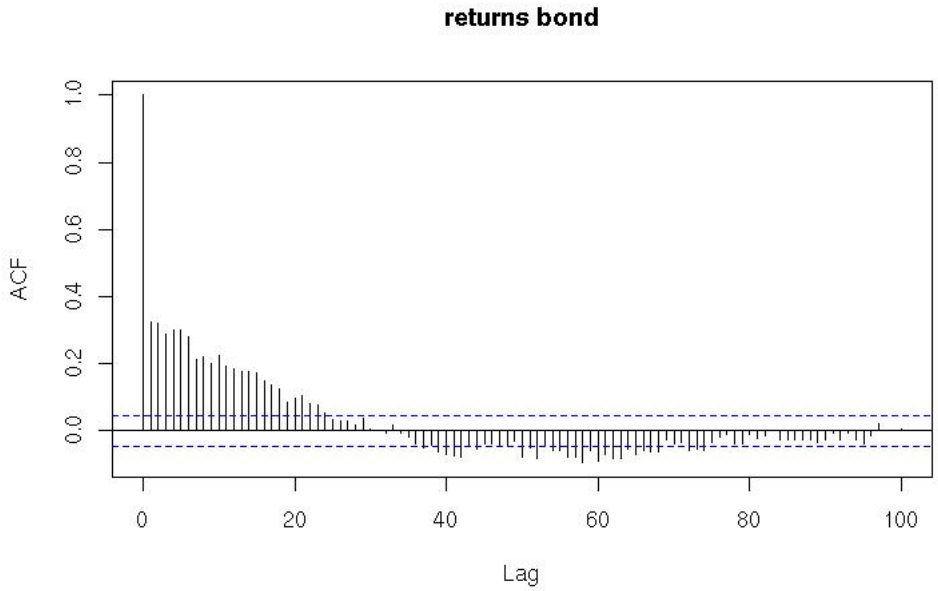

Figure 9: Autocorrelation of returns of the Bond

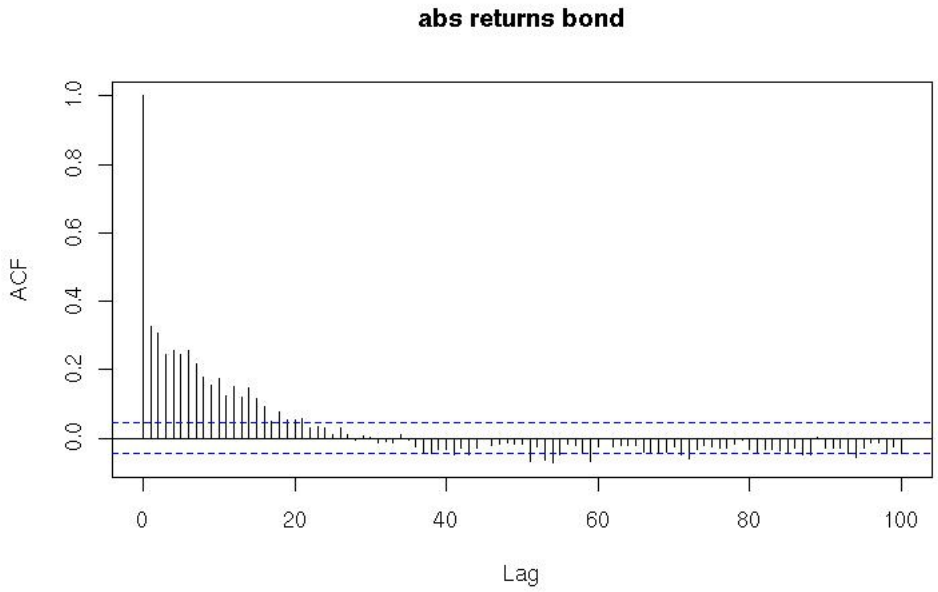

Figure 10: Autocorrelation of absolute returns of the Bond

From a statistical perspective, the reported results show the typical *"stylized facts"* of price time series in real markets, that is apparent geometric random walk of prices, *"fat tail"* of return distribution, and characteristic behavior of return and absolute return autocorrelation functions.

These results validate, at least statistically, the soundness of the market model and its implementation.

6 Conclusions

In this report we described the FLAME implementation of the Artificial Financial Market (AFM) developed in Eurace project, and demonstrated its use in several simulations.

During and after the development process, it's fundamental to verify that the system satisfies the requested specifications and provides the expected functionalities.

At first, we used "Unit testing" process to validate each transition function of the AFM system; then, in order to check the operation of all subsystems, or modules working together, we executed a "Functional testing" on the complete AFM system.

We ran the isolated Artificial Financial Market model and we conducted a statistical analysis to check whether the model is able to reproduce the main stylized facts found in time series of the stocks prices and bonds prices.

The results obtained correspond to those expected, and the system shown its ability to perform the required functionalities.

Consequently, the validation test has been passed, and we can trust thet the system is working correctly.

References

[1] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2007.

[2] Jean-Philippe Bouchaud. Power-laws in economics and finance: some ideas from physics. Science & Finance (CFM) working paper archive 500023, Science & Finance, Capital Fund Management, August 2000.

[3] Cont. Scaling and correlation in financial data. May 1997.

[4] Rama Cont. Empirical properties of asset returns: stylized facts and statistical issues. quantitative finance. *Quantitative Finance*, 1:223–236, 2001.

[5] J. Doyne Farmer. Physicists attempt to scale the ivory towers of finance. *Computing in Science and Engg.*, 1(6):26–39, 1999.

[6] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, New York, NY, USA, 1974. ACM.

[7] Brett McLaughlin, Gary Pollice, and David West. *Head First Object-Oriented Analysis and Design*. Addison-Wesley, 2007.

[8] Adrian Pagan. The econometrics of financial markets. *Journal of Empirical Finance*, 3(1):15–102, May 1996.

[9] Robert G. Sargent. Verification and validation of simulation models. In *WSC '07: Proceedings of the 39th conference on Winter simulation*, pages 124–137, Piscataway, NJ, USA, 2007. IEEE Press.

[10] UG. D6.1: Agent based models of financial markets. 2007.

[11] UNICA. D1.2: Agile methodologies for defining and testing agent-based models. 2007.

[12] USFD. D1.1: X-agent framework and software environment for agent-based models in economics. 2007.