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
D1.1: X-Agent framework and software environment for agent-based models in economics
Due date of deliverable:

Actual submission date:

Start date of project: September 1st 2006 Duration: 36 months

Organisation name of lead contractor for this deliverable
University of Sheffield - USFD

Revision 2
10/09/07

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

List of Tables

Executive Summary

Agent-based modelling provides more innovative approaches to facilitating research into the unresolved issues of complex systems. EURACE aims to use agent-based modelling to explore the fields of economics to model the European economy consequently providing insights into economic models, behaviour of human societies, better computational models and improving parallel computing paradigms. This document represents the Deliverable 1.1 which gives insights into the modelling framework, FLAME, and how it has been applied to economic modelling.

An agent-based modelling framework, FLAME [6], previously developed at the University of Sheffield, has been successfully used to model biological systems and uncovered useful results. The framework, which uses x-machines as the basic computational model is flexible enough to be applied to various disciplines from biology to economics. Some of the features which make it flexible have been described in the report:

- The framework uses XMML, X machine markup modelling language, to define agents and the communications between them.
- Various feature are provided by XMML and the framework which allows the modellers to easily use the framework to design their own models and test their outcomes.
- A few examples of its application to economic models have proven its success and have been presented here.
- The unit at the University of Sheffield (USFD) has closely been working with the other economic partners in gathering the requirements for system design of the economic models involved in EURACE. Documents produced by GREQAM[1] present abstract details of the economic requirements the design of the final model should contain. All of these issues have been targeted and translated into computational modelling terms.
- USFD has also been working with the unit STFC[2] to produce efficient models for deploying the agents onto parallel platforms producing platform independent and efficient parallel solutions to how various economic models will be brought together and communication hazards will be handled.

Various examples of economic models have been produced highlighting the success of the framework and XMML. These include the labour market and the credit market(work done at the Bielefeld working meeting (29/05/07 - 02/02/07)). Results of the labour market have been shown whereas the credit market is currently under construction.

This report presents the framework and the flexibility of how the XMML schema can be used to produce various models of the economy allowing

[1] Université de la Méditeranée.
[2] Rutherford Appleton Laboratories.

different units to design and test their models and bring them together into one simulation.

1 Introduction

This document contains the details of efforts to implement economic models as agent-based simulations.

The remainder of the document will be organised as follows:

- **Background** - Overview of agent-based modelling and software system specification;
- **Design Decisions** - Contains implementation issues surrounding the modelling requirements;
- **Framework Implementation** - Contains implementation details of the framework;
- **Model Creation** - Contains details about how to implement models;
- **Understanding Economic Models: The C@S Model** - Contains details on implementing the C@TS model;
- **Building Eurace by Markets** - Contains details on implementing the labour and credit markets;
- **Appendix A – XMML Schema**, which formally defines the XMML language;
- **Appendix B – C@TS Model XMML**, which formally defines the C@TS model;
- **Appendix C – Labour Market Model XMML**, which formally defines the labour market model;

This report presents the work completed for deliverable D1.1 depicting how the x-agent framework FLAME facilitates use of agent based modelling in economics. This deliverable acts as part of the workpackage 1 which comprises of agent-based software engineering methodologies being laid down for the project EURACE.

Keeping in accordance with Milestone 1.1, the report presents a definition of the XMML modelling language and how it is used for economics and how agent-based models of economics can be written using it.

2 Background

Agent-based modelling is a large research field allowing researchers to explore complex systems. Examples of which include ant and bee colonies, biological cellular structures and human societies. The importance of this approach is that it allows a bottom-up procedure, where the focus goes into the individual interacting units which possess defined rules. Accompanying these rules, when simulated, the individual interactions will produce an emergent pattern of behaviour which can be observed of the system as whole. This pattern can then be studied to test and understand the behaviour of the complex system deducing if the rules introduced were justifiable or need alteration. This helps deeper understanding of the interacting agents and their behaviour which was otherwise not easily observable if these systems were viewed as a whole.

The term '*agent*', as Tesfatsion [14] describes, 'refers broadly to a bundle of data and behavioural methods representing an entity constituting part of a computationally constructed world'. In economics, the definition of an agent can although vary from representing a group of agents like a firm composed of many individuals or an individual itself like a customer or a worker.

Agent-based modelling takes the view that systems can be modelled using many interacting objects. Objects, or agents, are self-contained autonomous machines that can communicate with each other. To put a more precise definition onto an agent, we suggest a formal computational model based on specifying software systems called X-machines. XMML is the modelling language used to represent these agents as X-machines and how they will be communicating between each other.

2.1 X-Machines

The X-machine is a general computational model introduced by Eilenberg [7] and later modified to represent more complex architectures at the University of Sheffield [8]. Contrary to Turing machines, X-machines have been used to model complex systems and have enhanced their own capability to more complex structures. One of the enhancements of the X-machine is the communicating X-machine of which there are several approaches [1, 2]. The approach used in XMML consists of a set of autonomous X-machines which use messages to communicate with each other. There are no explicit input or output components of these machines apart from this. Figure 2 depicts the structure of an X-machine agent.

Stream X-machines, introduced by Laycock [12], are another extension of the basic X-machine model and forms the basis for defining the agents in XMML. The basic definition of an agent would thus, in accordance to the computational model, contain the following components:

1. A finite set of internal states.

2. A set of transition functions that operate between states.

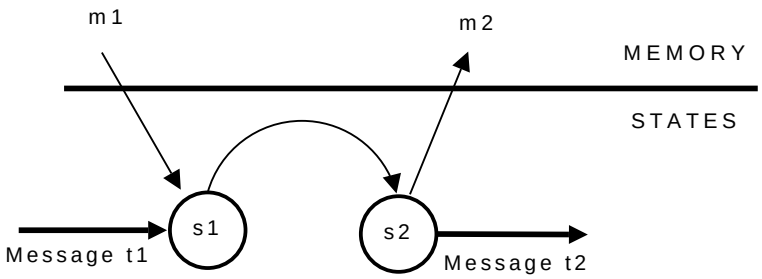

Figure 1: Transition function

3. An internal memory set. In practice, the memory would be a finite set and can be structured in any way required.

4. A language for sending and receiving messages between other agents.

$$X = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0) \tag{1}$$

where,

- Σ are the set of input alphabets
- Γ are the set of output alphabets
- Q denotes the set of states
- M denotes the variables in the memory. This can have a possibility of being infinite
- Φ denotes the set of partial functions ϕ that map and input and memory variable to an output and a change on the memory variable. The set ϕ: $\Sigma \times M \longrightarrow \Gamma \times M$
- F in the next state transition function, $F : Q \times \phi \longrightarrow Q$
- q_0 is the initial state and m_0 in the initial memory of the machine.

2.1.1 Transition Function

The transition functions allow the agents to change the state in which they are in, modifying their behaviour accordingly. These would require as inputs their current state s_1, current memory value m_1, and the possible arrival of a message that the agent is able to read, t_1. Depending on these three values the agent can then change to another state s_2, updates the memory to m_2 and optionally sends a message, t_2. Figure 1 depicts how the transition function works within the agent.

Some of the transition functions may not depend on the incoming message. Thus the message would then be represented as:

$$Message = \{\emptyset, < data >\} \tag{2}$$

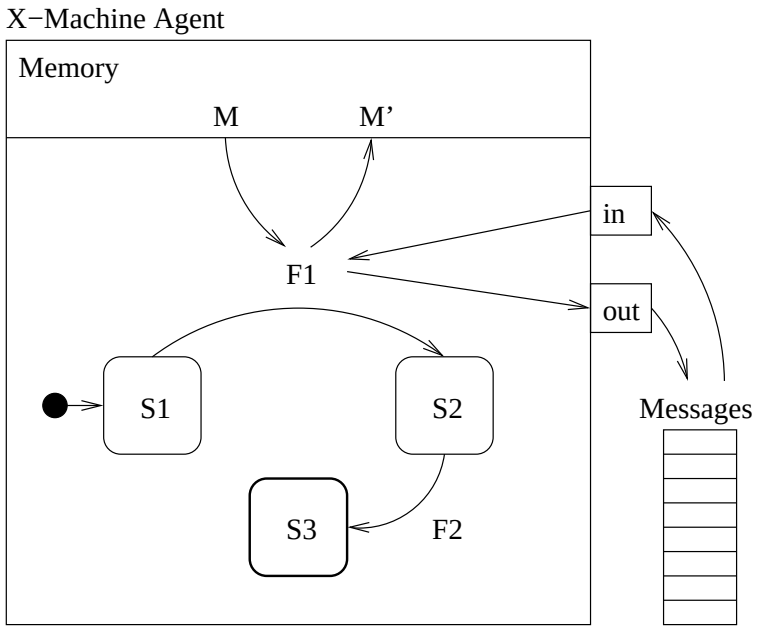

Figure 2: X-machine agent

These agent transition functions may be expressed in terms of stochastic rules, thus allowing the multi-agent systems to be termed as stochastic systems.

2.1.2 Memory and States

The difference between the internal set of states and the internal memory set allows for added flexibility when modelling systems. There can be agents with one internal state and all the complexity defined in the memory or equivalently, there could be agents with a trivial memory with the complexity then bound up in a large state space. There are good examples of choosing an appropriate balance between these two as this enables the complexity of the models to be better managed.

3 Design Decisions

After discussions with economists about an economic model for Europe, partners at the Université de la Méditerranée (GREQAM) created a modelling requirements and specifications document [16, 17]. This chapter describes the implementation issues surrounding these requirements/ specifications including formally specifying agents, transforming the specification into a simulation and the parallel processing issues of running a simulation on high performance parallel computers.

3.1 Feature Identification

The requirements document highlights the following issues for building high-fidelity, high-resolution agent-based models as described in [13]:

- Identify actors.
- Develop a set of operations that the actors perform.
- Define the applicable operations in a logical sequence.
- Identify and quantify the resources on-hand and remotely accessible to the actors.

3.2 System Description

Specifying software behaviour have traditionally involved finite state machines which allow modelling a system in terms of its inputs and outputs. More abstract system descriptions include UML which has already been used to design agent-based models[4, 3, 10, 19] but these techniques lack precise descriptions needed for generating simulation code and for testing. Testing a system specified as a finite state machine makes it easier for the behaviour to be expressed as a graph and allow traversals of all possible and impossible executions of the system [3]. Conventional state machines describe the state-dependent behaviour of a system in terms of its inputs, but this fails to include the effect of data. X-machines are an extension to conventional state machines that include the manipulation of memory as part of the system behaviour, and thus are a suitable way to specify agents. The advantages of this approach have been highlighted in Section 2.1. Describing a system would thus include the following individual stages for creating a model:

- Identifying the system functions
- Identify the states which impose some order of function execution
- Identify the input messages and output messages
- For each state identify the memory as the set of variables that are accessed by outgoing and incoming transition functions

[3] This is similar to branch traversal testing.

3.3 Labour Market Case Study

A text based specification of the labour market was created by partners at the University of Bielefeld [15], which defined two types of agent, Firms and Households. After Discussions at the working meeting in Bielefeld (29 May – 2 June 2007) the labour market algorithm can be summarised as follows:

1. Every month Firms calculate their production, including required workers
2. Firms act accordingly and send out any vacancies
3. Households receive vacancies, rank them, and send job applications
4. Firms receive applications, rank them, and send job offers
5. Households receive job offers, then send a offer acceptance
6. Firms receive offer acceptance(s) then update their wage offer (dependent on how many vacancies that are filled)

The sequence of operations described are meant to cover one working day in the simulation.

Following the method for creating an X-machine model, the firm agent system functions can start to be identified:

- Calculate production
- Send vacancies
- Receive applications
- Rank applications
- Send job offers
- Receive offer acceptance(s)
- Update wage offer

Also the system states that impose some order of function execution can start to be defined. This is achieved by associating transition functions with a start state and an end state, hence the transition between states (the start and finish state can be the same state), see Table 1. The next stage is to identify the input and output messages associated with a function transition, see Table 2. Finally identifying the pre and post memory of the transition functions, see Table 3. The same method can be applied to the Household agent, see Table 4.

Start State	Function	End State
producing	calculate production	prepare production
prepare production	send vacancies	get applications
get applications	receive an application	get applications
get applications	rank applications	applications ranked
applications ranked	send job offers	get offer acceptances
get offer acceptances	receive offer acceptances	get offer acceptances
get offer acceptances	update wage offer	producing

Table 1: Firm system states

Start State	Input	Function	End State	Output
producing	day of the month	calculate production	prepare production	
prepare production		send vacancies	get applications	vacancies
get applications	job application	receive an application	get applications	
get applications		rank applications	applications ranked	
applications ranked		send job offers	get offer acceptances	job offers
get offer acceptances	offer acceptance	receive offer acceptance	get offer acceptances	
get offer acceptances		update wage offer	producing	

Table 2: Firm input and output messages

Start State	M_{pre}	Input	Function	End State	M_{post}	Output
producing	day of month to act = x	day of the month = x	calculate production	prepare production	required_workers = calc_production()	
prepare production	required_workers > current_workers		send vacancies	get applications		vacancies
get applications		job application	receive applications	get applications		
get applications			rank applications	applications ranked		
applications ranked			send job offers	get offer acceptances		job offers
get offer acceptances		offer acceptance	receive offer acceptances	get offer acceptances	current_workers++	
get offer acceptances			update wage offer	producing	wage_offer = update_wage_offer()	

Table 3: Firm memory pre and post state transitions

Start State	M_{pre}	Input	Function	End State	M_{post}	Output
unemployed		vacancy	receive vacancies	unemployed		
unemployed			send applications	get offers		applications
get offers		offer	accept offer	employed		offer acceptance

Table 4: Household functions

For both the Firm and the Household a state transition diagram can be produced, see Figures 3 and 4. Immediately we can see that there is a missing state transition in the Household agent from employed to unemployed. To make this transition you would expect a redundancy message to arrive for the Household. This can be added to the state transitions as a function called 'made redundant' with start state 'employed', input 'redundancy', and end state 'unemployed'. The redundancy message must come from the employees firm, so we need to add this to the Firm agent. From the Firm state 'prepare production' there is only one transition function when 'required workers > current workers'. But there are two other instances when both values are equal or required workers is less than the current number of workers. In this case the firm would need to sack the appropriate number and send out redundancy messages. The final model is described by the Tables 5 and 6 with the state transition diagrams in Figures 5 and 6.

In some states where messages are being received, 'get_applications', 'get_offer_acceptance', and 'get_offers', there comes a point when the agent needs to stop waiting for incoming messages and perform some operation, like ranking. For the X-machine model any state transition requires an incoming message or the memory being in a required state. The memory state could include a count for the number of messages read and stop after a certain number. Except there could be the possibility of no incoming messages and therefore never reach the limiting value. Or a memory value could include an internal clock ticker and the agent waits for a certain amount of clock ticks, except there would need to be a mechanism to advance the clock tick. For a message event approach an incoming message could come from a central control agent that knows that there are no more messages to be read. This could be achieved by all agents that have finished sending a certain type of message, sending a message to the control agent. The control agent has a list of all agents that send the type of message and knows when they have all finished, then sends a message to agents that read in that type of message to say that no more messages are being sent. A final concept is that of a null message, or one that states that there are no more messages to read. This has been defined by the message types 'vacancies_finished', 'applications_finished', and 'offer_acceptance_finished' in the model.

The need for a null message is tied to the idea that the economic models are defined by a sequence of actions that must take place in one iteration, or working day. For example the labour market is run completely once every day. Therefore every agent needs to have available all incoming messages for the sequence to complete properly. Another view is that agents should not wait for all incoming messages as communication should be continuous, as should the labour market, as real labour markets do not start and complete on the same working day usually but is a continuous process over every working day. This strategy though could involve asynchronous updates, as described in Subsection 3.5 which would not be very compatible with parallel processing.

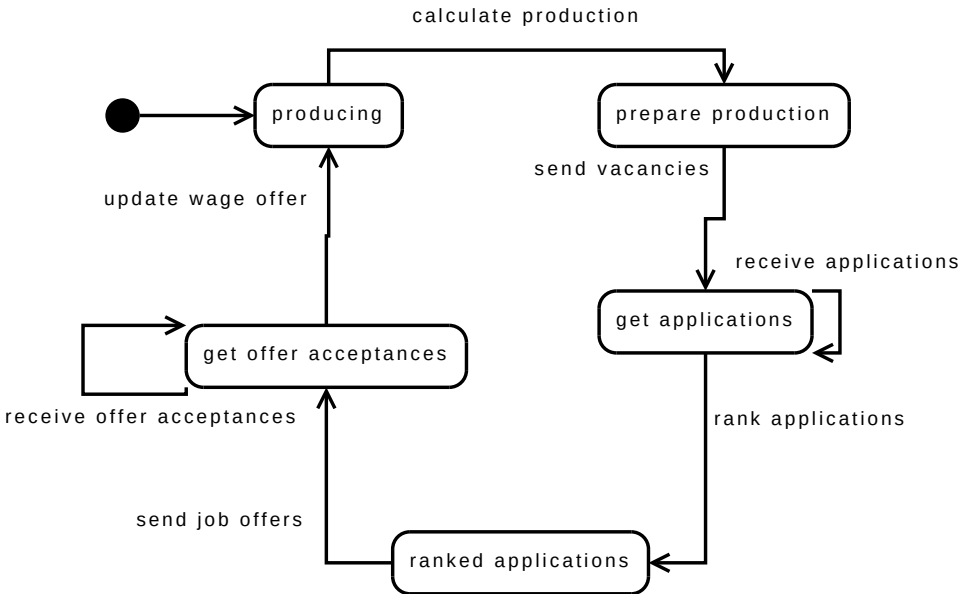

Figure 3: Firm state transition diagram

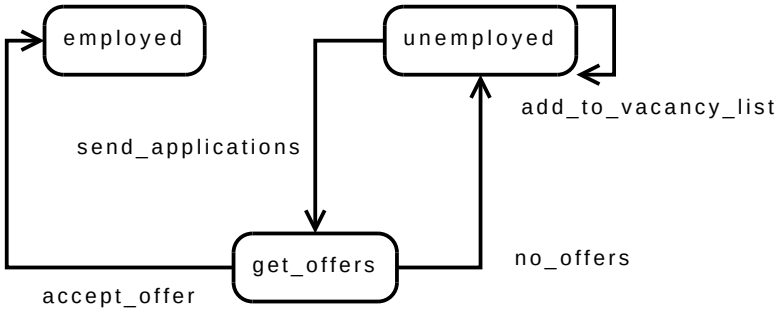

Figure 4: Household state transition diagram

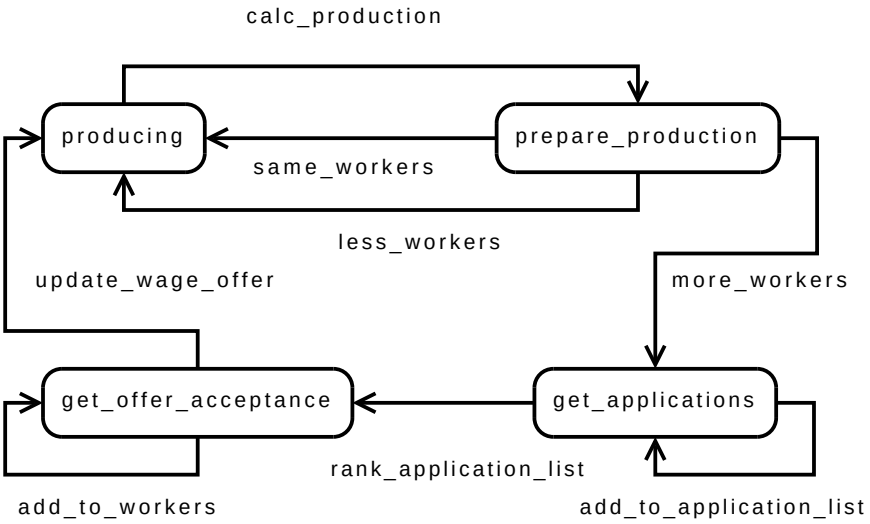

Figure 5: Firm state transition diagram updated

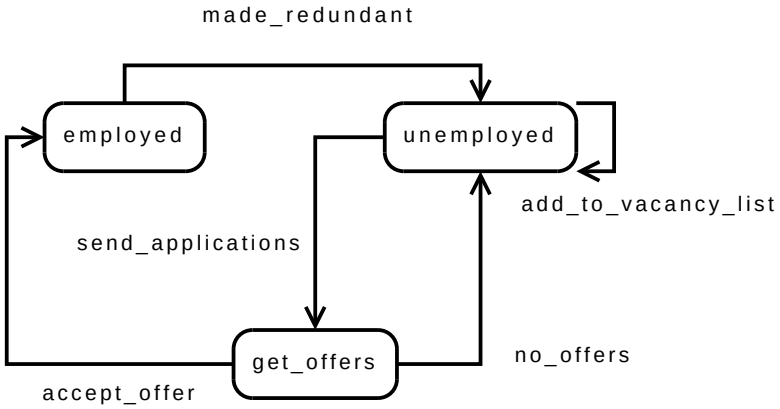

Figure 6: Household state transition diagram updated

State	M_{pre}	Input	Function	State	M_{post}	Output
producing	day_to_act = x	day_of_month(x)	calc_production	prepare_prod	required_workers = calc_production()	
prepare_prod	required_workers == current_workers		same_workers	producing		
prepare_prod	required_workers < current_workers		less_workers	producing	current_workers = required_workers	redundancies
prepare_prod	required_workers > current_workers		more_workers	get_applications		vacancies
get_applications		application	add_to_ application_list	get_applications		
get_applications		applications_finished	rank_application_list	get_offer_accept		job_offers
get_offer_accept		offer_accept	add_to_workers	get_offer_accept	current_workers++	
get_offer_accept		offer_accept_finished	update_wage_offer	producing		

Table 5: Firm functions

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		

Table 6: Household functions

3.4 Unified Modelling Framework

By creating a unified modelling framework partners on the project can use their expertise to create models of their own particular economic markets. These markets should then be able to be combined to create a macroscopic model of the European economy in a synergetic way. The unified modelling framework should also enable the parallel processing of a simulation independently from the model and its modellers.

Abstraction layers are very important as a way of hiding implementation details of a particular set of functionalities. Discussions with the STFC Rutherford Appleton Laboratory (STFC) have produced the following three layered approach. First the model layer that modellers interact with and have knowledge about. The perception at this level is of a collection of agents, that run through operations in order, and communicate. The second layer, the framework layer, is the engine of the simulation. It handles the reading in of agent start states, allocates agents to processors, runs agent operations in order, and sends agent messages. The third and final layer is the communication layer and handles agents receiving messages. Usually agents only read a relevant subset of all the messages sent, depending on various factors, and it is this layer that filters and subdivides the available messages. A block diagram of this approach has been presented in Figure 7.

3.5 Handling Of Time

Computer simulations operate on two notions of time:

- The advancement of processing time
- The advancement of simulation time

The processing time is the program progress and simulation time depends on program progress. For agent-based simulations processing time is the processing of agents and the handling of communication. Simulation time is advanced between periods of processing, for example when every agent is updated and all communication has reached its destination.

Deciding which agent to run and when to process/update it is a major issue.

For some theoretical results it can make a major difference in the outcome. The most dramatic example is the Game of Life where synchronous updates create patterns and structures capable of computation, but under an asynchronous scheme the world quickly becomes lifeless. Another example comes from game theory where synchronous turns of players can evolve oscillation of states while asynchronous player turns quickly find a stable equilibrium [9].

Particularly for communicating agents is when communication completes, which is when messages are sent when are they available to be received. This can involve two kinds of update strategies - synchronous, at the same time, and asynchronous, not at the same time. These updates can be defined in the context of communication as follows:

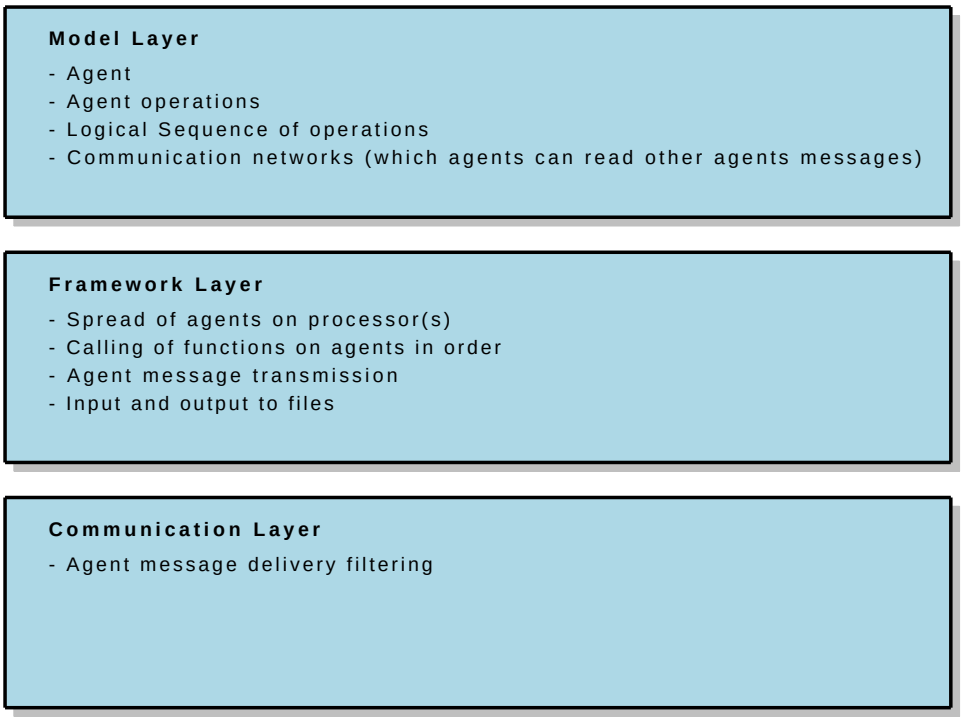

Figure 7: Layers of abstraction for the framework.

- Synchronous:
  - Communication only completes after every agent is updated once.
  - Order of agent updates does not matter.
- Asynchronous:
  - Communication completes after every agent is updated.
  - Order of agent updates matters.

3.5.1 Communication

Communication is very important when dealing with parallel processing of simulations. It can act as a major bottleneck that can slow down simulation times. Discussions with partners at STFC suggested that it is the starting up and ending of communication between processors that is the major factor and not the amount of data being sent [11]. This suggests that the least amount of communication synchronisation points, or completes, the better. It also implies that it is better to send as much information as possible in a single communication than to send each piece of communication individually.

Deciding which computer platform to be used should not affect the results of a simulation. Processing and communication time should affect simulation time and not in the other direction. So the framework should be designed to be platform independent. This becomes important when

handling agent updating and communication. In a simulation agent communication should not be affected by the number of processors used nor the physical networks connecting them [4]. Summarising the points to be considered:

It should not matter that an agent is not on the same computing node. This requires all agent interaction is achieved via contactless communication via messages. Contactless here refers to the inability to directly poll or access another agents memory values, as this is not possible if the agents are on a different computing nodes.

Any communication sent should be available for when it is needed to be read. This means operations that receive messages can only be run when messages have arrived. The physical bandwidth of the communication hardware used to run a simulation will not affect the results.

3.5.2 Updating Agents

There are two ways an agent can be updated/processed. Updating can be based on processing time information, called incremental based, or rely on incoming communication, called event based. Though incremental based self updating can include incoming communication, and event based could include an incoming timed event.

Because agents only communicate via messages, they can be updated at any time if any messages they need to read have arrived. So the only thing affecting the updating of agents is the communication dependencies, i.e. we can't update this agent until other agents have been updated. By using the state machine description to calculate the possible order of the functions, which shall be called internal dependencies, and the communication input and output between different functions, the communication dependencies, a function dependence graph can be created. A paper [18] from 2002 uses this dependence analysis technique to aid automated test case generation, which could also aid testing of models in the framework. Figure 8 shows a dependency graph for the labour market of all the actions that can happen in one day, i.e. after a date event happens and waiting for the next one. From the communication dependencies defined in the graph, one can add stages where communication must complete before the corresponding function requiring the input is processed. One can also assert that an agent can be updated until it is waiting for incoming communication and can only be updated again till after the corresponding communication completes. The graph also shows what agents need updated when, and depending what state they are in, the function that is executed.

[4] The speed of the cables or buses used for connection between processors responsible for carrying agent communication

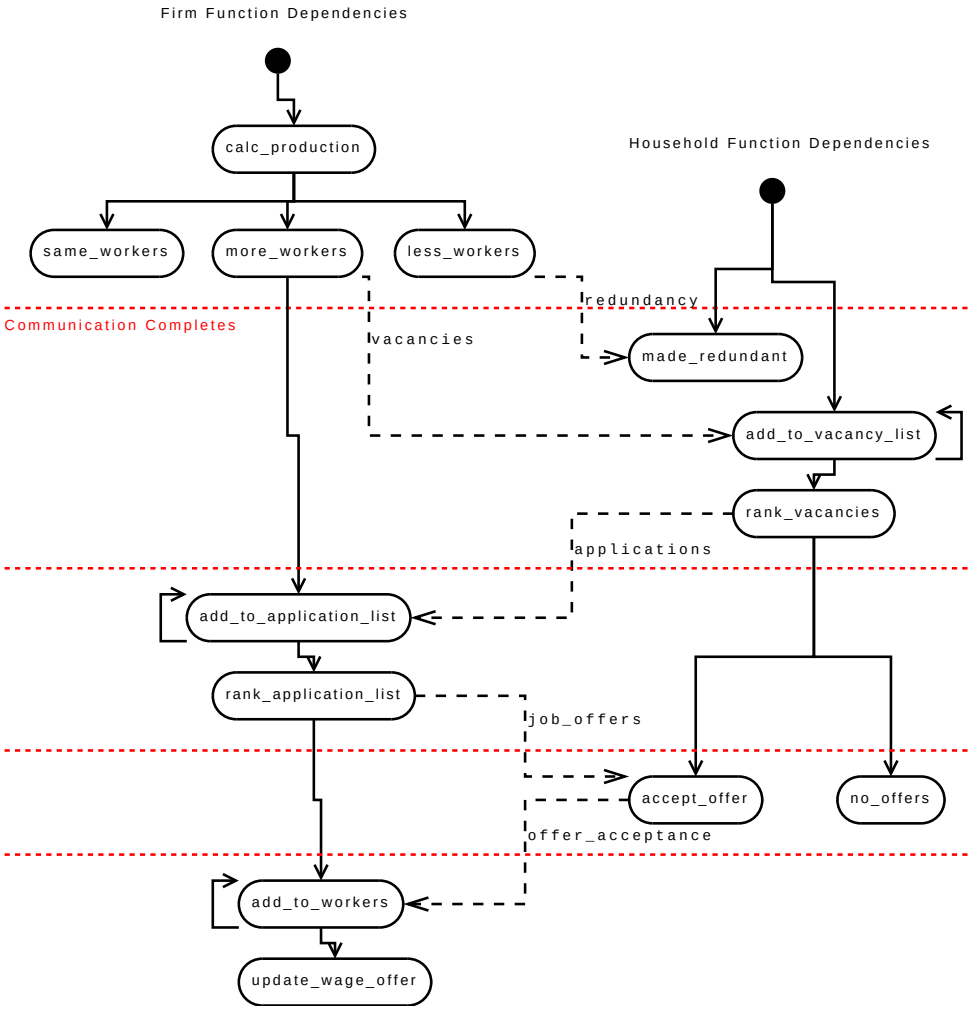

Figure 8: Labour market function dependencies

3.6 Communication Networks

Parallel computation is easily handled when agents are communicating via messages. The use of the idea of agent-agent and agent-environment interactions is an abstraction above the fundamentals. The only availability for agents communication are sending messages and receiving messages.

3.6.1 Agent-Environment Interaction

The idea of an 'environment' can be something that holds information that could possibly change, which can be embodied as an agent itself. Examples of environments in agent-based models can be:

- Land that grows crops (the ground cover environment).
- Chemical signals (the chemical environment).
- Newspaper business sections (the economic environment).

FLAME has been used for modelling biological systems, especially biological cells, where external solvers are needed to solve chemical diffusion and the physical movement of the cells. It is functions in these 'environment' agents that can be used to call external solvers, and pass back information back to the cells.

3.6.2 Agent-Agent Interaction

Agent-agent interaction is when one agent sends a message and another reads it. The agent reading messages can filter messages depending on specified variables. Examples of which include:

- Its 'id' (direct).
- Its 'region' (local area interaction).

Agents do not need to hold a list of pointers to other agents to represent their local neighbourhood. This can be achieved by the following ways:

- Agents having the same region number.
- Agents having a trade group number.
- Agents having a location and filtering messages via a distance metric.

Few instances, where the buyer has a preferred seller, such information would be held within the agent memory. Networks in agent based models are fully defined with agents, not a top down global view.

3.7 Simulation Output and Data Storage

Data storage is an important issue. Currently data is being held in XML format for ease of access but this presents problems with increasingly large file sizes. Other options to resolve this issue are being considered:

- Common Data Format (CDF) for the storage and manipulation of multi-dimensional data sets
- Database which would also easier extraction of specific data
- XML alternatives: YAML, JSON, SDL

[add recent work by Shawn and Susheel?]

4 Framework Implementation

Initial work on implementation had already been undertaken by Simon Coakley as part of his Ph.D. This involved creating a parser program that takes a model description as an input and produces a runnable simulation program, either in serial or parallel. Model descriptions are written in a file format called XMML which is a specific tag defined XML file. The XML format provides a structure for data that computers and humans can understand. A model description file allows metadata about a model to be used to direct source code creation (via a parser program), especially for parallel code that modellers do not need to encounter. It can also be used to direct testing efforts and produce diagrams of a model that aid in its understanding.

4.1 Xparser

The Xparser is the name of the program that reads XMML model files and produces simulation program code, see Figure 9. Additional features that have been added since the project started include:

- Function dependencies – agent functions can now be ordered in such a way that the simulation program can execute them at the best possible moment (which is calculated), and allows for future use of threading techniques.
- Template engine – the logic behind the generated simulation code has been transfered to template files so that collaboration between partners is easier.
- Dynamic arrays for agent memory – the ability to have dynamic sized arrays in agent memory has been added (although movement of agents on a parallel machine used for load balancing has yet to be implemented).

The Xparser also has an XML reader to read the XMML model descriptions, and also generates graphs of the function dependencies for analysis.

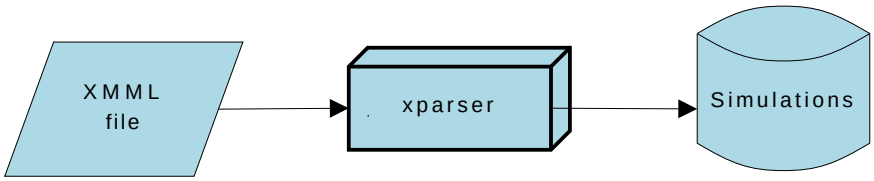

Figure 9: Xparser usage

The Xparser is completely written in C with the use of standard libraries only. This was so that the program could be deployed on any platform (with a C compiler) simply and easily. Because most of the logic is held in the simulation template files it is viable to create a program in any language or use additional libraries that would do the same job as the Xparser.

4.1.1 Process Sequence

Agent functionality is defined by its functions. Functions change the agent state and drive a simulation forward. The sequence that these functions are run is determined by their dependency on each other, defined in the model XMML. Dependencies are either communication, dependent on messages, or internal, dependent on agent internal memory.

It is possible to construct a dependency graph (a directed acyclic graph) to show the sequence of events that happen in a simulation. Whenever a communication dependency occurs, in parallel, this requires a synchronisation block between the nodes so that messages arrive in time to meet the dependency. These synchronisation blocks are a major time bottle neck and so the fewer there are the more efficient the simulation. By traversing a dependency graph it is possible to calculate the most efficient time to run functions and where best to place synchronisation blocks.

Creating the function dependency graph currently uses a simple algorithm. It finds functions with no dependencies on it, assigns them a layer, removes them from the graph, and reruns the algorithm.

Figure 10 shows eight functions with dependencies. All are communication (denoted with a 'C') except the dependency of Function 2 on Function 5 which is internal (denoted with an 'I'). Because internal dependencies do not need a communication synchronisation block we can organise the synchronisation blocks in such a way that we need the least amount of them. An example of this strategy is the organisation of the functions from Figure 10 into layers separated by synchronisation blocks in Figure 11.

4.2 Framework Communication

The usual attribute that separates agent-based models from other modelling techniques (like differential equations) is the use of space. Agents have a location attribute that places them in space in relation to other agents. To create new results from this added dimension of space, communication is usually restricted to a distance metric, so that information is kept localised. This knowledge can be used to direct efficient communication in a model implementation.

Currently to efficiently handle messages with respect to localised communication: The current implementation of the framework is based around the idea of space as a Cartesian scale in 1, 2 or 3 dimensions, with:

- All agents defined with a Cartesian location
- All messages are defined with originating Cartesian location and range
- Agent space is partitioned along Cartesian lines

In this way when a message is sent by an agent, the message can be defined as originating from the agent location and can only be read by agents with location that is defined within the message range. To aid efficiency messages are only sent to partitions in agent space that include agents within the message range. After discussions with members of the STFC unit about

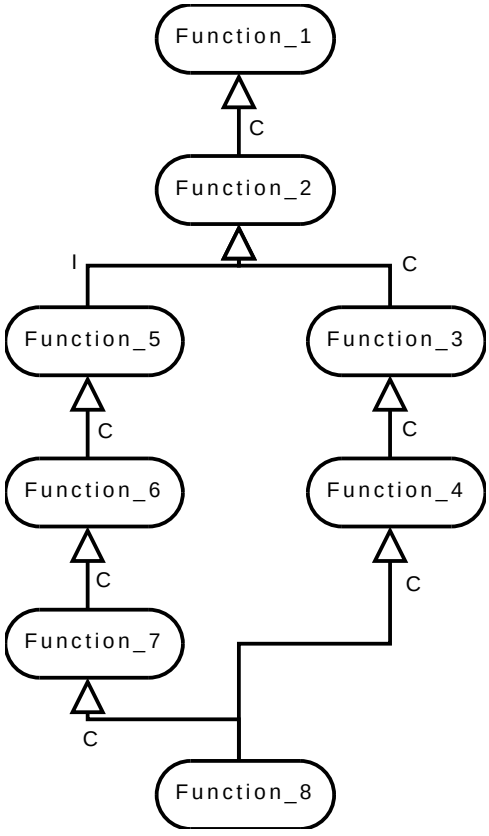

Figure 10: Communication dependencies between functions

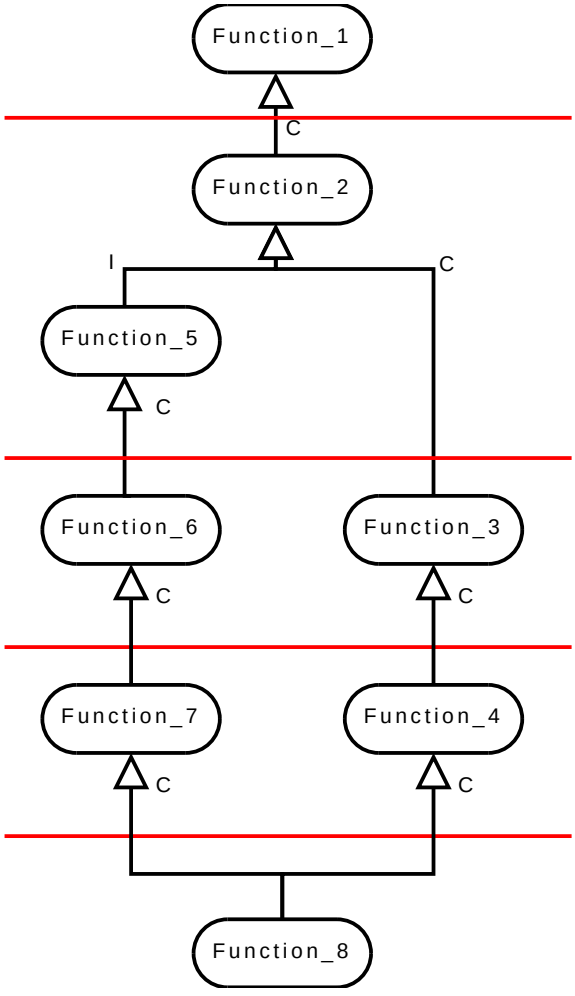

Figure 11: Syncing communication dependencies as synchronisation layers

parallel communication in HPCs the filtering of messages that are to be sent to different nodes is not required. Firstly that filtering of messages is done twice, when messages are sent between nodes, and when agents try and read incoming messages. Secondly that the filtering of messages before they are sent between nodes is unnecessary. This is because the time cost of sending messages between nodes is more weighted on the opening and closing of communication and less on the actual amount of data that is sent [11], this iterates the importance of keeping communication synchronisation blocks to a minimum. Therefore it is more efficient to send all out going communication to all nodes. This then shifts all efficiency efforts onto the filtering of messages for agents to read. This strategy is mentioned in Section 3.4.

Also in efforts to make the framework more generic the idea of space should not be restricted by a Cartesian scale, or in fact any distance scale. This is because agent space might be defined as groupings, for example NUTS-2 regions.

4.3 X Machine Agent Markup Modelling Language (XMML)

A description language for agent-based simulations, XMML has been presented here. XMML is orientated towards representing agent-based models as formalised abstract state machines, particularly communicating X-machines. The motivation was to provide a formalised framework to enhance creating and testing of agent-based models and also provide innate parallel processing capabilities.

4.3.1 Features of XMML

There are a number of factors which make XMML unique to achieve its research purposes. A few have been listed below:

- XMML is not restricted by research area.
- It is not restricted by any grid or location based structure.
- Communication is not restricted between agents, but mechanisms are available to efficiently filter incoming messages.
- Agents are updated at the most efficient time and in parallel (if available)

XMML is meant to aid agent-based modellers in developing more formalised models that are easier to create, test, share, and be parallel processable without additional work. The definition of the model description language here does not specify how to parse the model description into a simulation program but defines what is required and how the simulation is advanced.

4.3.2 Data

Variables represent the data that is possessed by the agent in their memory and the messages they send or receive. While executing a simulation program the details of this data needs to be known in advance. The advantage of this approach are that data structures and algorithms that handle data, especially in parallel, can be automatically generated:

- Creating data structures for agents and messages
- Creating functions that handle input and output to files
- Creating functions that access agent memory
- Creating functions that interact with messages
- Creating parallel algorithms that handle data between nodes

Variables contain a data type and a name. Data types are used to assign storage for the variable and define the type of data that will be held in that location. Variable names are used to reference and alter the data if needed. The following XML represents a variable of type *float* and named *temperature*.

```
<var>
  <type>float</type>
  <name>temperature</name>
</var>
```

4.3.3 C Language

The current XMML to simulation code parser is written in the C programming language, therefore allowing C data types to be used. Examples of these have been given in Table 7.

Type	Description	Usual Byte Size	Example Usage
int	Integer number	2 bytes	int count; count = 5;
float	A single-precision floating point value	4 bytes	float temp; temp = 6.2;
double	A double-precision floating point value	8 bytes	double sun_temp; sun_temp = 13600000.0;
char	Character	1 byte	char letter; letter= 'a';

Table 7: C fundamental data types.

```
<datatype>
<name>employee</name>
<desc>Used to hold employee information</desc>
<var><type>int</type><name>id</name></var>
<var><type>float</type><name>wage</name></var>
</datatype>
```

Table 8: Example of the employee data type.

4.3.4 Data Structures

To facilitate more structured data representation, new custom data types can also be created. These custom data types can allow C data types as well, and they can be referred to by their own user defined names. Table 8 gives an example of a custom data type called *employee* which holds an 'id' of type *int* and a 'wage' of type *float*.

The <desc> </desc> tags can be used to allows users to describe the data type which can later be extracted to be used for description in the documentation. These custom data types can now be used in the same way as the C data types.

4.3.5 Array

Variables can also be defined as a list which can also be represented as an array. The array can either be static, with predefined size, or dynamic, allowing its size to change. To define a static array, use the C syntax which is to place square brackets after the variable name that contains the array size. So for a list of six variables of type float called wage, the definition would be (Table 9):

```
<var><type>float</type><name>wage[6]</name></var>
```

Table 9: Defining an array of predefined size.

Dynamic arrays have their own special data type provided by the XMML. For any data type name just add '_array' at the end. Therefore to change the static array above to a dynamic array, take away the square brackets and size and add '_array' to the data type name (Table 10):

```
<var><type>float_array</type><name>wage</name></var>
```

Table 10: Defining a dynamic array.

4.3.6 XMML Components

XMML components are the representation of how models are described in its specification. The description comprises of the agents involved, the agent characteristics and the messages being used to communicate among the agents.

```
<!−−******* X-machine Agent - Firm **********−−>
<xmachine>
<name>Firm</name>
<!−−−−−−−−−−Variables−−−−−−−−−−−−−>
<!−− All variables used by Firm are declared
here to allocate them in memory −−>
<memory>
<var><type>int</type><name>id</name></var>
<var><type>double</type><name>value</name></var>
</memory>
<!−−−−−−−−−−−− Defining functions −−−−−−−−>
<functions>
<function><name>Firm_1</name></function>
<function><name>Firm_3</name></function>
</functions>
</xmachine>
```

Table 11: Example of a Firm Agent.

```
<messages>
<!−−Message for stock of the firm−−−−−>
<message>
<name>firm_stock</name>
<note>This message lets the people know how much stock
the firm they are buying from has left.</note>
<var><type>int</type><name>firm_id</name></var>
<var><type>int</type><name>stock</name></var>
</message>
</messages>
```

Table 12: Example of describing messages.

4.3.7 Agents

Every agent is a x-machine. This depicts that the agent would thus contain a set of memory variables which it can update during its functions. The agent would also have a set of functions it can perform. The actual function definition is not part of XMML component and is defined separately in a C file. Table 11 gives an example of a firm agent.

4.3.8 Messages

Messages are used to communicate between the agents. All messages are enclosed in the <messages> </messages> tag and every message structure is defined seperately. An example has been shown in Table 12.

5 Model Creation

5.1 Data structures

From the definitions in model XMML data structures can be created for agent memory and message memory.

Agent and message memory is made up of variables of certain data types. These can be:

- C fundamental data types - int, float, double, char (Table 7).
- Abstract data types made up of more than one C data type.
- Static arrays of C data types and abstract data types.
- Dynamic arrays of C data types and abstract data types.

Dynamic arrays are a built-in feature of the framework (for sending messages in parallel the size of the array is needed). For any data type just add '_array' to the end, and access it via the following functions:

- datatype_array * my_array = init_datatype_array();

  For initialising the array.
- add_datatype(my_array, value);

  For adding an element to the array.
- remove_datatype(my_array, index);

  For removing element at the specified index.
- my_array->size;

  for returning the length of the array.
- free_datatype_array(my_array);

  For freeing the array.

5.2 Definition of XMML tags

The model description is given in the XML file using XMML tags which have been described previously. These tags are used by the xparser to recognise the agent memory, the sort of variables being used and the functions they can perform.

5.3 Handling Variables in Agent Memory

The xparser offers a few functions which can be used to access the variables in the agent memory.

- set_variablename(*value*)

The set function can be called with in the agent function to change the value of the variable in the memory. The following brackets contain the value to be replaced with.

- x=get_variablename()

The get function can be called within the agent function and gets the value of the variable wanted and saves it to the local x value.

5.4 Handling Messages

- add_messagename_message(*var1, var2,...*)

To add the message onto the message board. *Var1, var2* symbolize the value of the variables that the message carries.

- messagename_message=get_first_messagename_message()

The local variable gets the first message to traverse through the message.

- messagename_message->*var1*

The above command allows you to get the value of *var1* from the message.

- messagename_message = get_next_messagename_message(messagename_message);

The above command allows the loop to move onto the next message on the board to read through. This would be used with a while loop until it returns a *null*.

5.5 Handling Dynamic Arrays

The framework allows dynamic arrays to be used within the memory of the agent. This is useful when the agent needs to maintain a list of a continually growing nature of variables.

- int_array * Agents = init_int_array()

The above command initializes the dynamic array.

- xmachine_memory_agentname * xmemory = current_xmachine->xmachine_agentname;

To access the memory the xmemory pointer needs to be used with the current xmachine to point to the xmachine being accessed. The pointer would be of the type of the agent being accessed.

- reset_int_array(xmemory->dynamicvariablename);

When accessing the dynamic variable array we can use the reset to reset the array.

- add_int(xmemory->dynamicvariablename, messagename_message->var1);

To add to the dynamic array list use the above command with the name of the array given first and the value after the comma.

- xmemory->dynamicvariablename->array[value]

Values in the dynamic array can be accessed similar to the way elements in an array would be accessed.

- xmemory->dynamicvariablename->size

The size can be used to return the value of the size of the array. This would be changing continually as it is not fixed.

- free_int_array(agents);

To free the list of the agents used.

5.6 Handling Data Structures

5.7 Outputs Produced by the Xparser

5.7.1 Dotty graphs

Order of agent functions (creating a dependency graph from the function dependencies). Figures 12, 13 are examples of the outputs produced.

5.7.2 FireFox graphs

The svg files, produced after compilation with the xparser, allow a clear understanding on how the functions will be ordered during execution. Figure 14 depicts the output of the svg files. The red points depict a synchronization point, at which point the functions prior to it would have finished executing and sent out messages which the later functions can then read and proceed.

This figure also depicts how the functions will be distributed in a parallel manner. more than one function in the layer can be run on more nodes and all information could be brought together at the synchronization point.

6 Understanding Economic Models: The C@S Model

The first effort to create an economic model centred around the C@S project with papers in progress provided by the Ancona unit [5]. The model describes a sequential economy populated with large numbers of firms and workers/consumers who partake on markets for homogeneous non-storable consumption goods and labour services.

Newly introduced into the model was the idea of locality, at the heart of parallelising efforts. Where agents were given a location on a two dimensional continuous plane. The distance between agents in this Cartesian space affected if they could communicate with each other.

6.1 Version 1: Without the Mall Agent

From the paper describing the C@S model, two agents Firm and Person were implemented. Table 13 shows the relation between the event sequence described in the paper and the order of the agent functions. The importance of this version is that only one agent function is used when firms hire workers and persons buy goods. This is achievable only because these functions are run sequentially, and are therefore not parallelable. They depend on messages sent by the running of the function on other agents, and is therefore a self-dependency. The means the function needs to be run one after the other with messages sent available immediately to the consequent function run on other agents. The self-dependency is shown in the functions 'Firm_3' and 'Person_5' in the function dependency diagram in Figure 12.

6.2 Version 2: With the Mall Agent

The second version included a new agent type (defined by the XMML in Appendix B). The mall agent was introduced to parallelise the labour and goods markets and also to make the markets fairer as the cheapest workers and goods would be evenly distributed rather than first come take everything approach of version one. It also added locality of agents and the feature that firms and persons had to choose which mall to go to for the labour market and the goods market. The function dependency graph of this model is shown in Figure 13. Figure 14 shows the communication synchronisation points between the different functions of the agents. The red lines represent the points at which all functions prior to it would need to be finished for the simulation to proceed.

[More comment here possibly?]

6.2.1 Graphs

Graph 15 represents the pattern of behaviour between the goods sold, price and the production. when the price increase the goods sold reduces which causes a reduction in the production. Simultaneously if the price is low, more goods are sold causing more production to take place.

Graph 16 shows the relation of wages, price, production and goods sold. The price of the goods denotes a price inflation when the price has increased

Function (model)	Event (paper)	Firm Agent	Person Agent
1	1,2,3	Checks financial viability Calculates production Calculates labour required Send price inflation message	Does nothing
2	4	Does nothing	Calculate total price inflation (from price inflation messages) Calculate new wage Send job application messages
3	5	Hire workers (from job application messages and send hired messages)	Does nothing
	6		
4	7a	Calculate wage bill Calculate goods price (send price message) Calculate produced goods (send stock amount message)	Check hired messages (update status if hired)
5	7b	Does nothing	Spend income on goods (using firms price and stock messages, send updated stock messages if buying)
6	8	Calculate stock sold (from stock messages) Calculate revenue Calculate profits	Add wage to income (for next iteration)

Table 13: Sequence of events in the C@S model

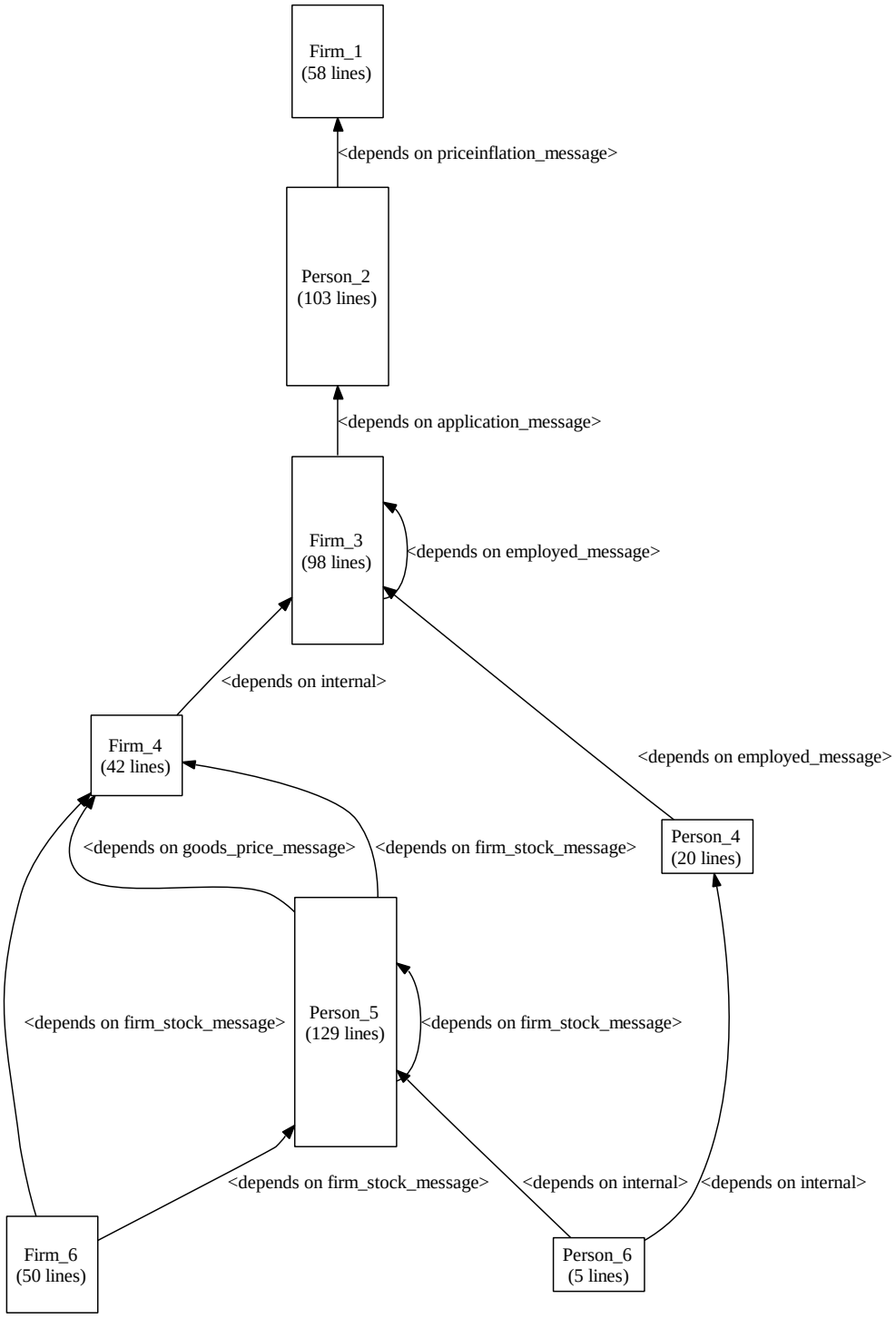

Figure 12: Function dependency graph of C@S model version 1

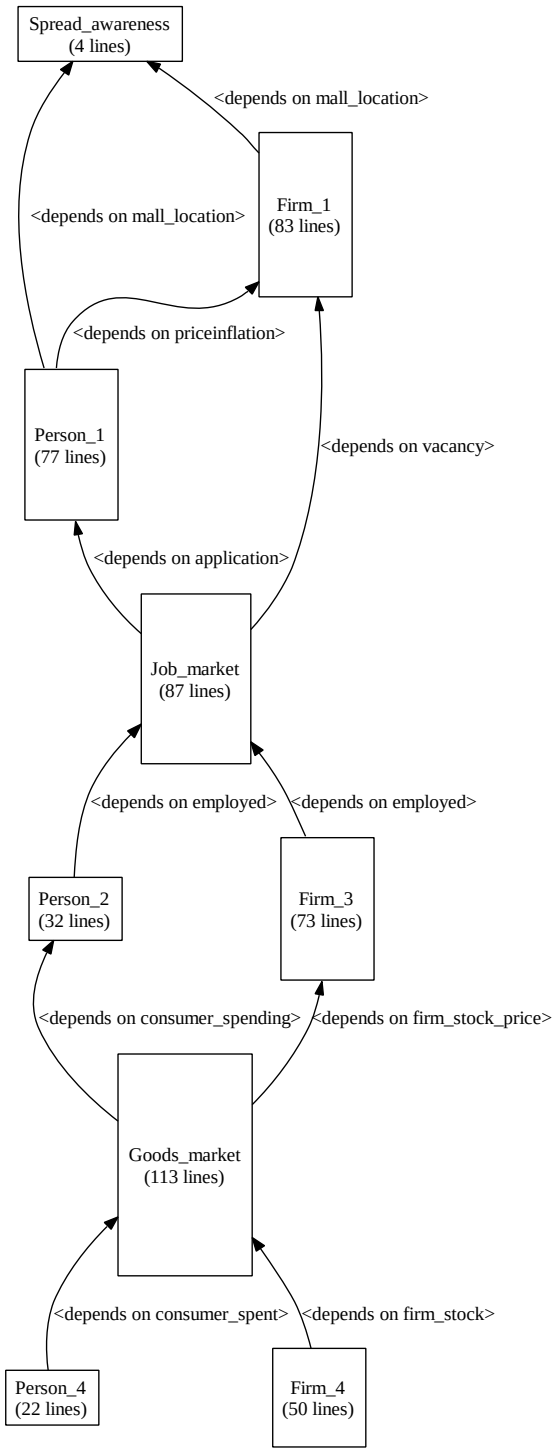

Figure 13: Function dependency graph of C@S model version 2

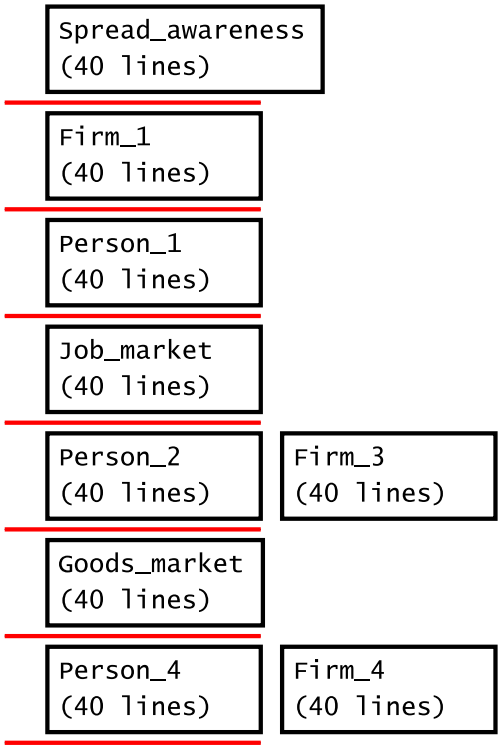

Figure 14: Communication synchronisation layers of C@S model version 2

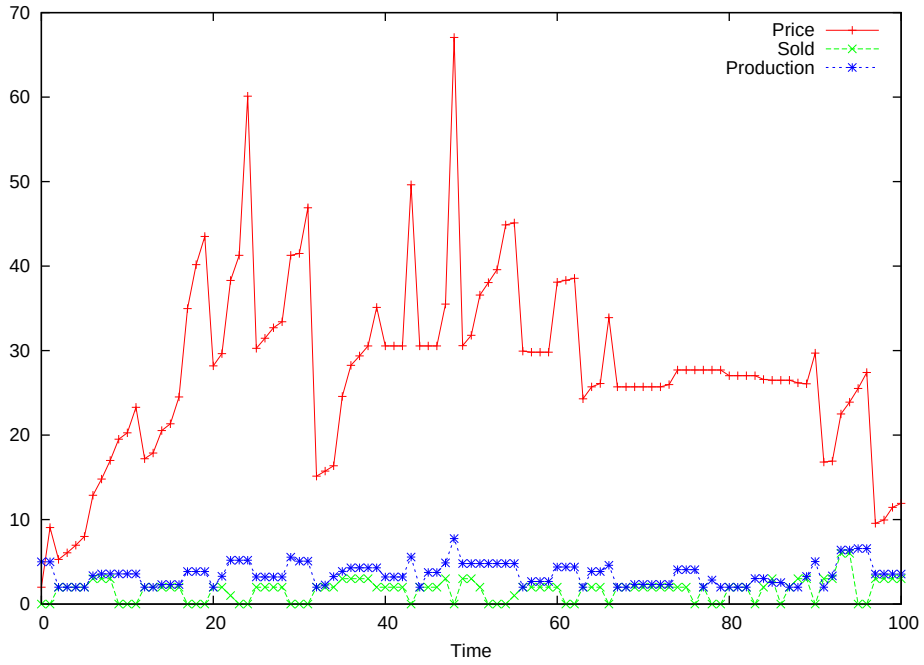

Figure 15: Graph showing the relation of price, stock sold, and production

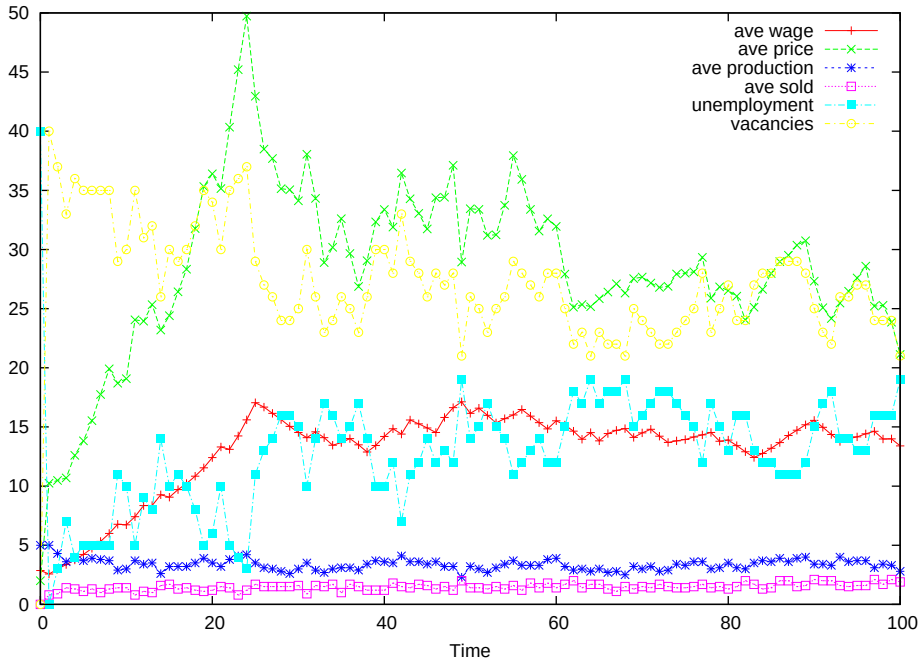

Figure 16: Graph showing the relation between average wage, price, production, and the stock sold

in the previous iteration. With an increase in price the wage of the workers increase as they demand higher wages. The goods sold and the production seem to have a mirror relation between them. When goods sold increase production increases and vice versa. The interesting facet about this relation is that the two seem to reach an equilibrium even if the are greater fluctuations in the wage and the price.

7 Building Eurace by Markets

7.1 The Labour Market Model

The labour market was initially described in a document by the Universitaet Bielefeld unit. This describes a matching algorithm between job seekers and vacancies in a six step approach. Initially each step was assigned per day in a simulation.

Step	Firm	Household
1	Send Vacancies	
2		Read Vacancies
3	Read applications, rank, and send job offers	
4		Read job offers,rank and send job acceptance
5	Read job acceptances	
6	Update wage offer	Update reservation wage

Table 14: Six step labour market algorithm.

At the Bielefeld working meeting the following were discussed:

- Communication between agents happen between agent functions:
  - One agent function sends a messages, and another agent function reads the message.
- When one function depends on a message sent from another function this is called a communication function dependency.
- When one function depends on the outcome of another functions within in the same agent this is called an internal function dependency.
- Function dependencies are not allowed to happen across time steps (days).
  - Because each day is taken to be a separate simulation run (this removes many problems).

As a group task the function dependencies of agent functions where discussed and written on a black board. This started to produce pairings of functions where communication needs to happen. These were written as arrows with a capital 'C' to denote a communication dependency. The six step algorithm then became four function communication dependency pairs:

After some discussions which included time scales, firm production frequency and overlap, it was decided that the whole labour market model would act completely every day. This allowed one function dependency graph to describe the total labour market algorithm. This was written up in XMML as a model description, see Appendix C.

Where each agent functionality was given a separate function 'box'. In fact if there is only an internal dependency between agent functions, both functions can be merged, as has been done for the first implemented version.

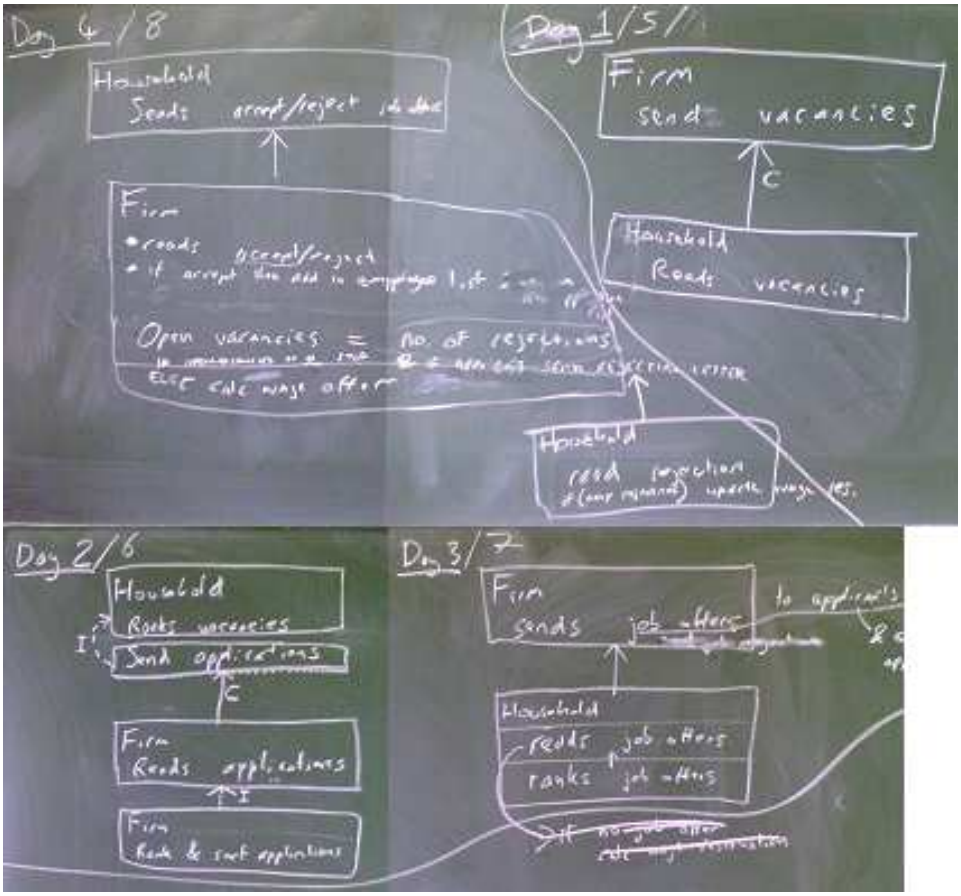

Figure 17: Blackboard diagrams describing discussions on the labour market model

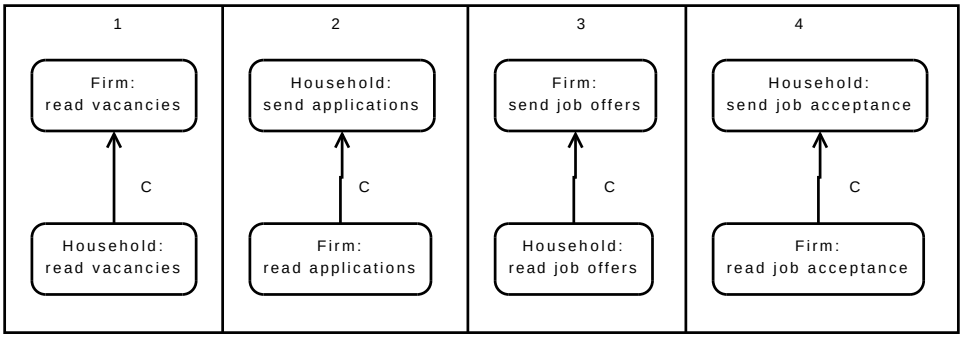

Figure 18: Function dependency of the labour market day by day

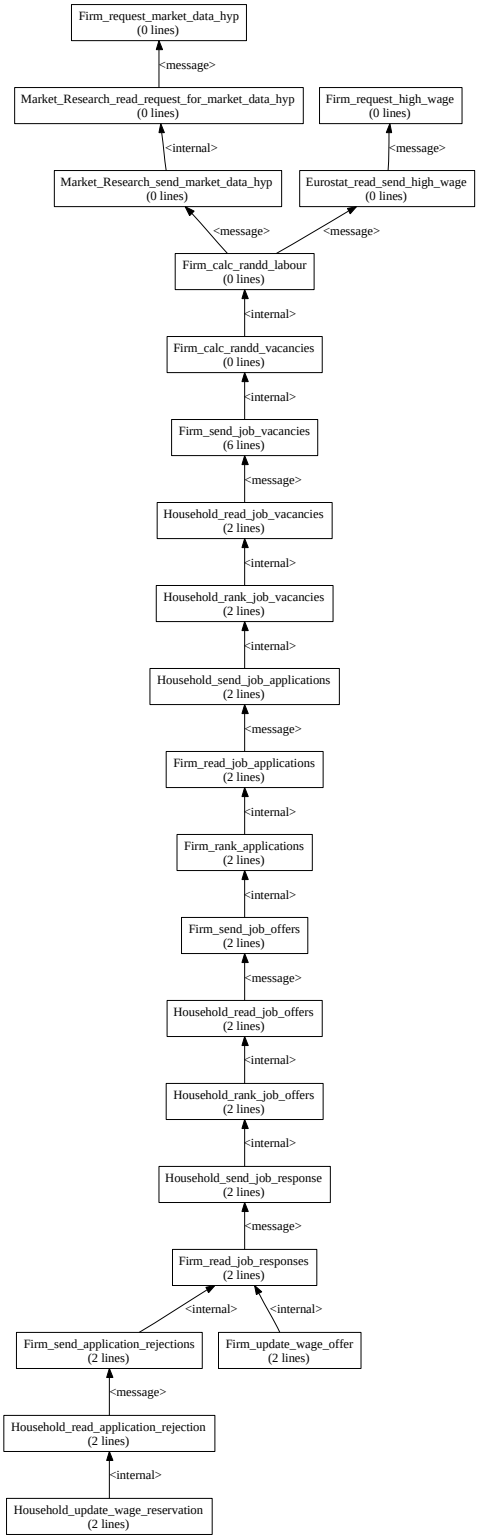

Figure 19: Function dependency graph of the labour market

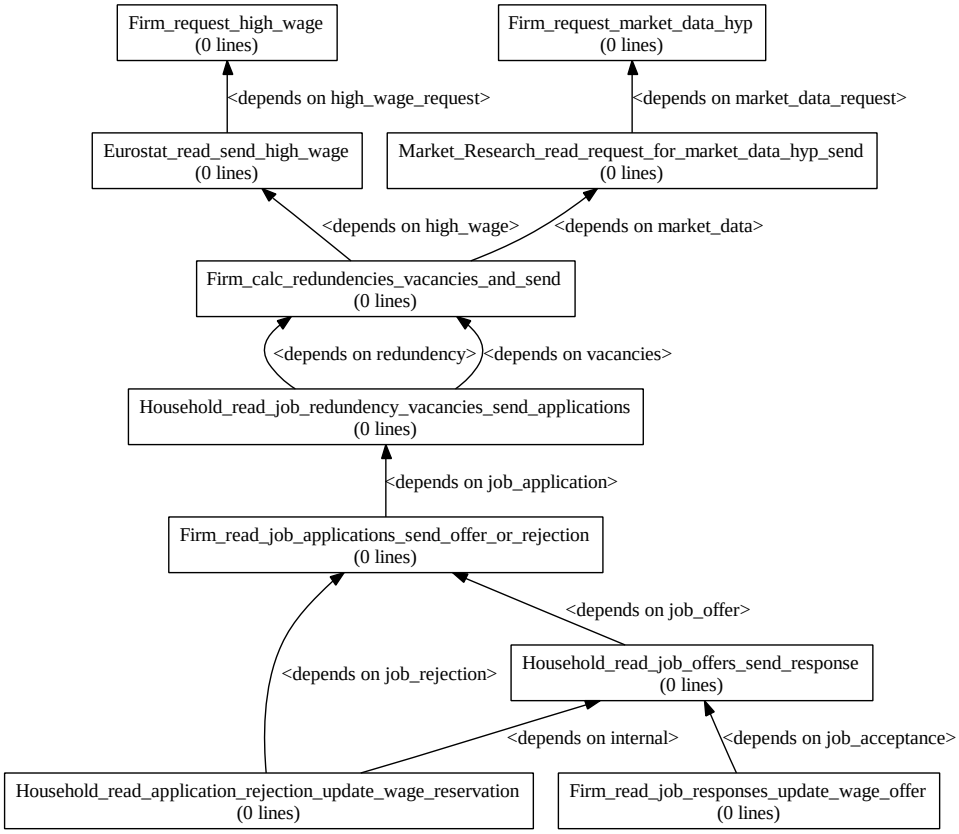

Figure 20: Function dependency graph of the labour market updated

7.1.1 Agents

The agent types used, their behaviour, and their number in the labour market are:

- Household - looking for a job - 1000
- Firm - looking to fill vacancies - 100
- Market Research - sends market data to firms - 1
- Eurostat - sends high wage information to firms - 1

Which corresponds to one region or NUTS-2 regions.

7.1.2 Function Layout

All internal function dependencies have been removed where they are the only dependency and the functions merged. This creates a more compact and easier to read model. It also saves memory space as creating lists (of vacancies, application, and job offers), ranking, and deleting them can all be done locally in the same function without saving the list to agent memory.

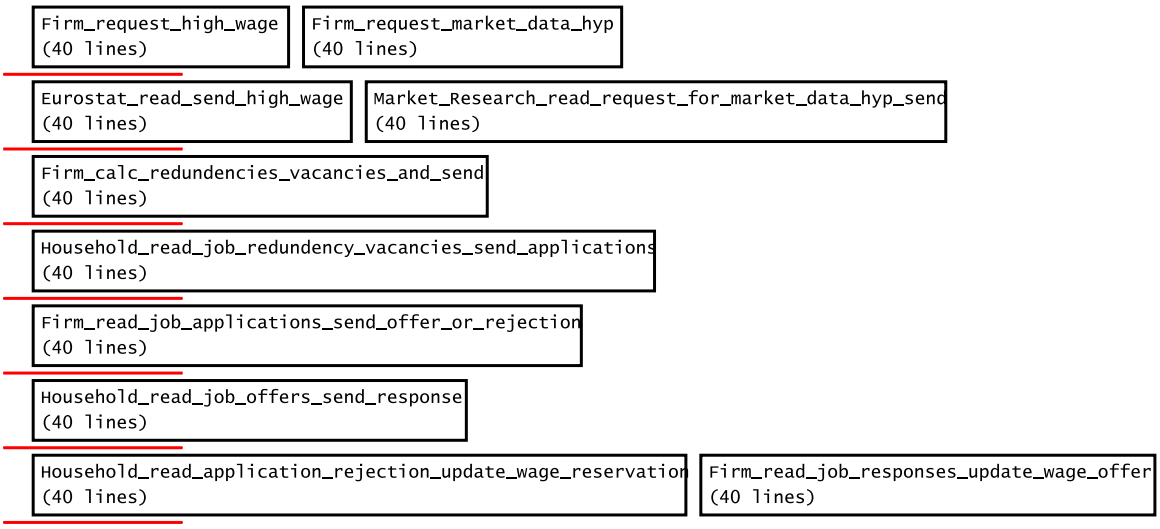

Figure 21: The communication sycnronisation layers of the labour market model

7.1.3 Implementation

The labour market revolves around the monthly cycle of firms calculating their monthly production. On the day of month they calculate this they ask the eurostat agent and market research agent for additional information then calculate the number of employees they need. If they have too many employees they send redundancy messages out, and if they have too few they send out vacancy messages.

7.1.4 Results and Conclusions

The labour market model provides a test bed to research:

- Ways to design models
  - Function dependencies
- Ways to implement models
  - Cluster internal functions
- Ways to run models efficiently

Running models efficiently includes calculating when best to run functions and where to place communication synchronisation points between functions. The image below lists the functions on rows with communication points as red lines. This is the order the functions will be run in with communication handled at the red lines. The main efficiency to be gained is to have as few amount of communication points as possible, as this is the main bottle neck in parallel (the starting up of communication between nodes).

As part of handling messages efficiently the message board will automatically organise messages in relation to the filters agents use to read messages, for example with a distance metric.

7.2 The Asset Market Model

Also discussed at the Bielefeld work meeting, the asset market is still under construction. Figures 23, 24 and 25 depict the block diagrams of the market and how it functions[5].

7.2.1 Agent Population

Household Agent. Invests in assets.

Firm Agent. Issues assets (stocks and bonds).

Financial Advisor Agent. Gives advice to households on the past performance of a set of asset allocation rules. It holds a database of such rules in its internal memory.

Asset Management Company. A firm that manages Exchange Traded Funds (ETFs) and/or hedge funds. Like other firms, the Financial Advisor distributes its profits to its shareholders. There can be multiple Financial Advisors.

Clearinghouse Agent. Reads limit_order_messages. Computes transaction prices.

LimitOrderBook Agent. Reads limit_order_messages. Computes transaction prices.

7.2.2 Internal Dependencies

A few internal dependencies in the household and the firm agent have been identified. These have been depicted in the dotty diagram in Figure 22.

7.2.3 Implementation

The asset market works on a monthly basis as depicted in Figure 23. The diagram shows that it starts on the first day of the month and continues normal procedure for the rest of the month. The first column, however, depicts how the asset market connects to the markets outside its own realm, like the 'Consumption Goods Market (CGM)' and external agents as the banks and the firms. A few implementation issues were encountered as listed below:

Problem of *Days*. Figure 23 shows how some of the functions of the asset market have to be run on one day on the month with the rest of the functions running as normal.

This gives rise to the discussion on how long is one simulation supposed to be depicted for. To make it default, USFD has proposed to make the length of one simulation to depict one day.

[5]These figures are a result of the discussion at the Bielefeld work meeting and converted into figurative form by Sander van der Hoog from the GREQAM unit.

There is another issue on how the date of the day will be checked as some of the functions are dependent on which day of the month it is. Discussions with the STFC unit have led to a few conclusions of either including a date check in every function description or the presence of a date agent allowing some functions to be executed. Further tests will be done to find the most efficient manner of doing this.

Internal and message dependencies. Referring to Figures 24 and 25 depict the function dependencies to be of two types - internal or message. As previously discussed on communication dependencies, internal represents dependency on functions within the agent. Thus some of these functions can be combined into one. Combining functions allows possibly more efficient use of memory and a more readable function dependency graphs, but removes the possibility of parallel execution (by future use of threads) and the ease of testing smaller functions. The external message dependency , is when the agent depends on another agent for data. Therefore these would be the synchronisation point at which prior to it all agents would have finished working and wait to move into the next block. The importance of reducing the layers to message dependencies reduces the synchronisation points to be encountered. this reduces computational overhead in the model.

7.2.4 Current Work

The model has already been designed and implemented in MATLAB by the Genoa unit in collaboration with the GREQAM unit and is going to be converted into C language for implementation using the framework.

Figure 22: Dotty diagram of household and firms.

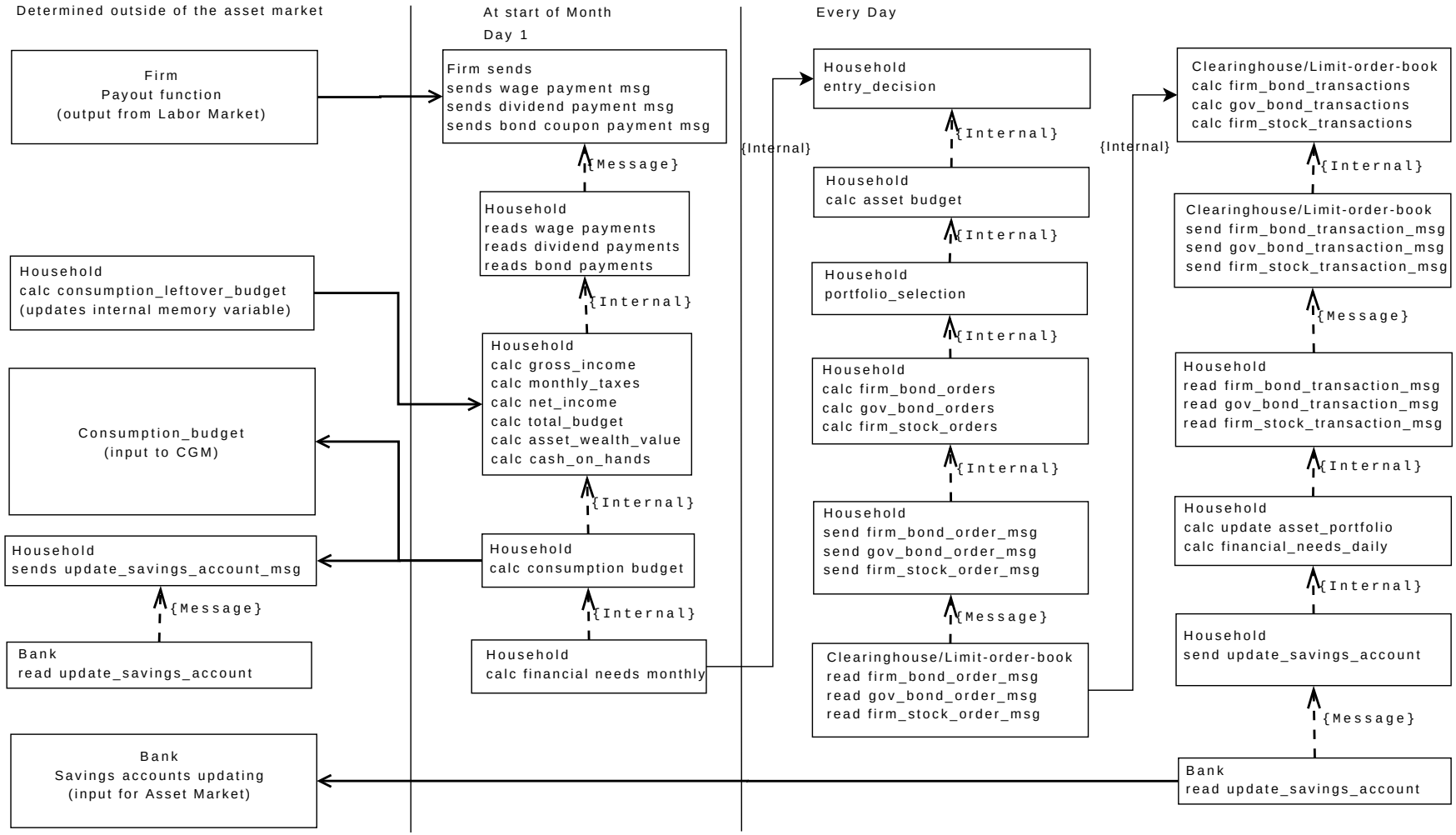

Figure 23: Function dependency graph for the Financial Management Role of the Household.

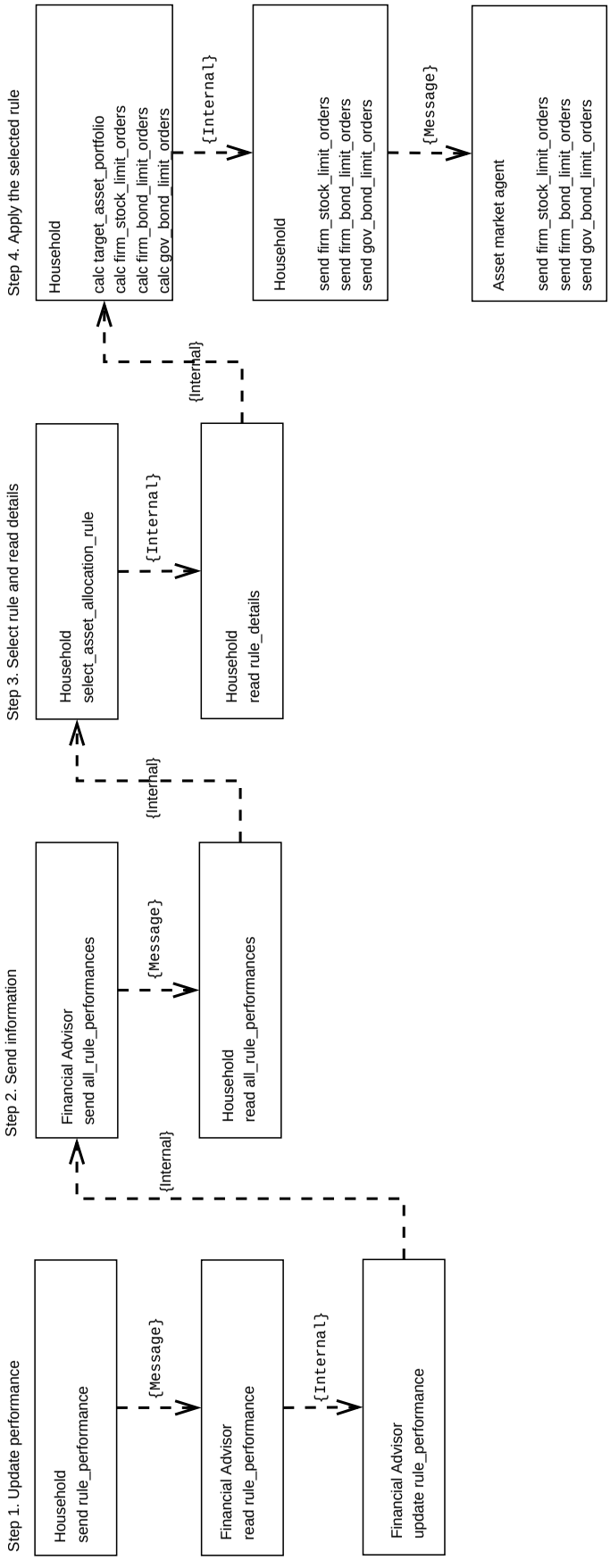

Figure 24: Function dependency graph for the Portfolio Selection Algorithm of the Household.

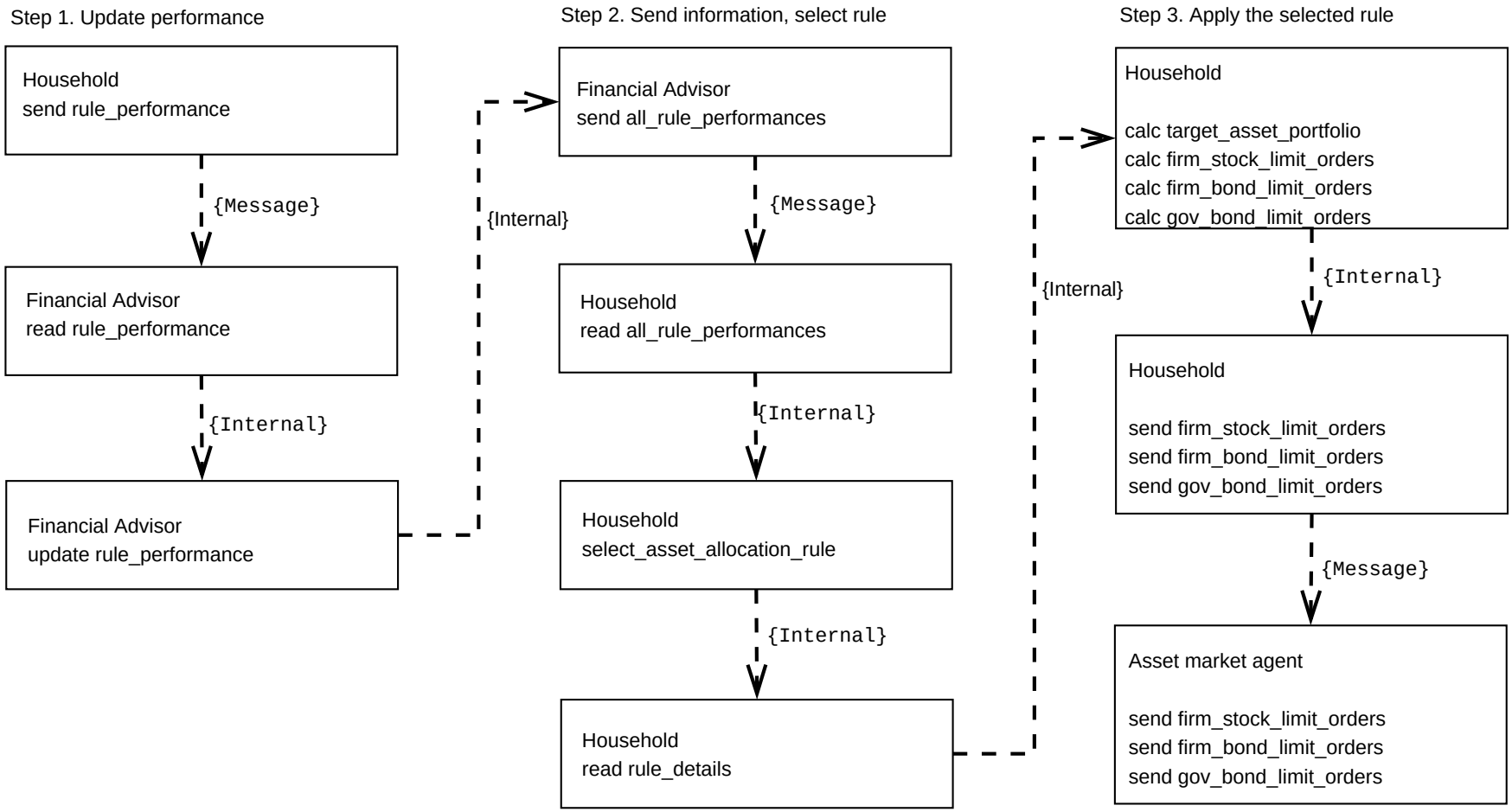

Figure 25: Function dependency graph for the Portfolio Selection Algorithm of the Household, showing the three communication layers.

A XMML Schema

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="xmachine_agent_model">
<xs:complexType>
 <xs:sequence>

  <xs:element name="name" type="xs:string"/>
  <xs:element name="author" type="xs:string"/>
  <xs:element name="date" type="xs:string"/>
  <xs:element name="notes" type="xs:string"/>

  <xs:element name="environment" minOccurs="0" maxOccurs="1">
   <xs:complexType>
    <xs:sequence>

     <xs:element name="constants" minOccurs="0" maxOccurs="1">
      <xs:complexType>
       <xs:sequence>

        <xs:element name="var" minOccurs="0">
         <xs:complexType>
          <xs:sequence>
           <xs:element name="type" type="xs:string"/>
           <xs:element name="name" type="xs:string"/>
          </xs:sequence>
         </xs:complexType>
        </xs:element>

       </xs:sequence>
      </xs:complexType>
     </xs:element>

     <xs:element name="functions" minOccurs="0" maxOccurs="1">
      <xs:complexType>
       <xs:sequence>

        <xs:element name="file" type="xs:string" minOccurs="1">
        </xs:element>

       </xs:sequence>
      </xs:complexType>
     </xs:element>

     <xs:element name="datatypes" minOccurs="0" maxOccurs="1">
      <xs:complexType>
       <xs:sequence>

        <xs:element name="datatype" minOccurs="1">
         <xs:complexType>
          <xs:sequence>
           <xs:element name="name" type="xs:string"/>
```

```
          <xs:element name="desc" type="xs:string"/>

          <xs:element name="var" minOccurs="1">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="type" type="xs:string"/>
             <xs:element name="name" type="xs:string"/>
            </xs:sequence>
           </xs:complexType>
          </xs:element>

         </xs:sequence>
        </xs:complexType>
       </xs:element>

       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>

   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="xmachine" minOccurs="1">
  <xs:complexType>
   <xs:sequence>

    <xs:element name="name" type="xs:string"/>

    <xs:element name="memory">
     <xs:complexType>
      <xs:sequence>

       <xs:element name="var" minOccurs="1">
        <xs:complexType>
         <xs:sequence>

          <xs:element name="type" type="xs:string"/>
          <xs:element name="name" type="xs:string"/>

         </xs:sequence>
        </xs:complexType>
       </xs:element>

      </xs:sequence>
     </xs:complexType>
    </xs:element>

    <xs:element name="functions" minOccurs="0" maxOccurs="1">
     <xs:complexType>
      <xs:sequence>

       <xs:element name="function" minOccurs="1">
```

```
        <xs:complexType>
         <xs:sequence>

          <xs:element name="name" type="xs:string"/>
          <xs:element name="depends" minOccurs="0">
           <xs:complexType>
            <xs:sequence>

             <xs:element name="name" type="xs:string"/>
             <xs:element name="type" type="xs:string"/>

            </xs:sequence>
           </xs:complexType>
          </xs:element>

         </xs:sequence>
        </xs:complexType>
       </xs:element>

     </xs:sequence>
    </xs:complexType>
   </xs:element>

  </xs:sequence>
 </xs:complexType>
</xs:element>

<xs:element name="messages">
 <xs:complexType>
  <xs:sequence>

   <xs:element name="message" minOccurs="1">
    <xs:complexType>
     <xs:sequence>

      <xs:element name="name" type="xs:string"/>
      <xs:element name="var" minOccurs="1">
       <xs:complexType>
        <xs:sequence>

         <xs:element name="type" type="xs:string"/>
         <xs:element name="name" type="xs:string"/>

        </xs:sequence>
       </xs:complexType>
      </xs:element>

     </xs:sequence>
    </xs:complexType>
   </xs:element>

  </xs:sequence>
 </xs:complexType>
</xs:element>
```

```
  <xs:element name="iteration_end_code" minOccurs="0">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="code" type="xs:string"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>

 </xs:sequence>
</xs:complexType>
</xs:element>

</xs:schema>
```

B C@TS Model

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xmachine_agent_model>
<name>C@S Bis Model</name>
<author>Simon Coakley and Mariam Kiran</author>
<date>011006</date>

<!--*****Environment values and functions *******-->
<environment>
<functions>
<file>functions.c</file>
</functions>
</environment>

<!--******* X-machine Agent - Firm ******-->
<xmachine>
<name>Firm</name>
<!--             Variables               -->
<!-- All variables used by Firm are declared here
to allocate them in memory -->
<memory>
<var><type>int</type><name>id</name></var>
<var><type>double</type><name>value</name></var>
<var><type>double</type><name>a</name></var>
<var><type>double</type><name>productivity</name></var>
<var><type>double</type><name>profits</name></var>
<var><type>double</type><name>f</name></var>
<var><type>double</type><name>production</name></var>
<var><type>int</type><name>goodsproduced</name></var>
<var><type>int</type><name>stock</name></var>
<var><type>int</type><name>sold</name></var>
<var><type>int</type><name>labour</name></var>
<var><type>int</type><name>numberofworkers</name></var>
<var><type>double</type><name>price</name></var>
<var><type>double</type><name>oldprice</name></var>
<var><type>double</type><name>priceinflation</name></var>
<var><type>double</type><name>sprice</name></var>
<var><type>double</type><name>lprice</name></var>
<var><type>int_array</type><name>workerid</name></var>
<var><type>double_array</type><name>workerwage</name></var>
<var><type>double</type><name>avewage</name></var>
<var><type>int_array</type><name>mall_id</name></var>
<var><type>int</type><name>mall_vacancy</name></var>
<var><type>int</type><name>mall_goods</name></var>
<var><type>double</type><name>posx</name></var>
<var><type>double</type><name>posy</name></var>
</memory>
<!--     Defining functions     -->
<functions>

<function><name>Firm_1</name>
<depends>
```

```
<name>Spread_awareness</name><type>mall_location</type>
</depends>
</function>

<function><name>Firm_3</name>
<depends>
<name>Job_market</name><type>employed</type>
</depends>
</function>

<function><name>Firm_4</name>
<depends>
<name>Goods_market</name><type>firm_stock</type>
</depends>
</function>

</functions>
</xmachine>
<!--********** End of Agent - Firm ***************-->

<!--******* X-machine Agent - Person ************-->
<xmachine>
<name>Person</name>
<!--      Variables for the Person        -->
<memory>
<var><type>int</type><name>id</name></var>
<var><type>double</type><name>savings</name></var>
<var><type>double</type><name>wage</name></var>
<var><type>int</type><name>firmid</name></var>
<var><type>int</type><name>mall_application</name></var>
<var><type>int</type><name>mall_shopping</name></var>
<var><type>int_array</type><name>mall_id</name></var>
<var><type>double</type><name>posx</name></var>
<var><type>double</type><name>posy</name></var>
</memory>
<!--        Defining functions               -->
<functions>

<function><name>Person_1</name>
<depends>
<name>Firm_1</name><type>priceinflation</type>
</depends>
<depends>
<name>Spread_awareness</name><type>mall_location</type>
</depends>
</function>

<function><name>Person_2</name>
<depends>
<name>Job_market</name><type>employed</type>
</depends>
</function>

<function><name>Person_4</name>
```

```
<depends>
<name>Goods_market</name><type>consumer_spent</type>
</depends>
</function>

</functions>
</xmachine>
<!--****** End of Agent - Person ************-->

<!--******* X-machine Agent - Mall **********-->
<xmachine>
<name>Mall</name>
<!--      Variables for the Mall        -->
<memory>
<var><type>int</type><name>id</name></var>
<var><type>int_array</type><name>app_person_ids</name></var>
<var><type>double_array</type><name>app_person_wages</name></var>
<var><type>int_array</type><name>sell_firm_ids</name></var>
<var><type>int_array</type><name>sell_firm_stocks</name></var>
<var><type>double</type><name>posx</name></var>
<var><type>double</type><name>posy</name></var>
</memory>
<!--        Defining functions               -->
<functions>

<function><name>Spread_awareness</name></function>

<function><name>Job_market</name>
<depends><name>Firm_1</name><type>vacancy</type></depends>
<depends><name>Person_1</name><type>application</type></depends>
</function>

<function><name>Goods_market</name>
<depends>
<name>Firm_3</name><type>firm_stock_price</type>
</depends>
<depends>
<name>Person_2</name><type>consumer_spending</type>
</depends>
</function>

</functions>
</xmachine>
<!--********** End of Agent - Mall   *************-->

<!--** Messages being posted by the agents to communicate **-->
<messages>
<!--  Message posted to record the price inflation    -->
<message>
<name>mall_location</name>
<note>Mall location message</note>
<var><type>int</type><name>mall_id</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
```

```
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>
<!--  Message posted to record the price inflation   -->
<message>
<name>priceinflation</name>
<note>This message is posted by the firm when it calculates the next price
of the goods. The message is read by the workers to help calculate their
new wages because they consider the price inflation to do this</note>
<var><type>int</type><name>firm_id</name></var>
<var><type>double</type><name>priceinflation</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>
<!--     Message for applying to firm      -->
<message>
<name>application</name>
<note>This message is posted by the worker that it is applying to this
firm for work with what wage he wants and where he worked before</note>
<var><type>int</type><name>person_id</name></var>
<var><type>double</type><name>person_wage</name></var>
<var><type>int</type><name>mall_id</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>
<!--     Message for firm vacancies      -->
<message>
<name>vacancy</name>
<note>Message for firm vacancies</note>
<var><type>int</type><name>firm_id</name></var>
<var><type>int</type><name>vacancies</name></var>
<var><type>int</type><name>mall_id</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>
<!--  Message that the person has been employed     -->
<message>
<name>employed</name>
<note>This message is sent out by the firms to let the workers
know who are employed and by whom</note>
<var><type>int</type><name>person_id</name></var>
<var><type>double</type><name>person_wage</name></var>
<var><type>int</type><name>firm_id</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>
```

```
<!--   Message for consumer spending   -->
<message>
<name>consumer_spending</name>
<note>Message to Mall outlet indicating how much to spend</note>
<var><type>int</type><name>person_id</name></var>
<var><type>double</type><name>spending</name></var>
<var><type>int</type><name>mall_id</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>
<!--  Message for consumer spent   -->
<message>
<name>consumer_spent</name>
<note>Message from Mall outlet indicating how much has been spent</note>
<var><type>int</type><name>person_id</name></var>
<var><type>double</type><name>spent</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>
<!--  Message for stock of the firm  -->
<message>
<name>firm_stock</name>
<note>This message lets the people know how much stock the firm
they are buying from has left.</note>
<var><type>int</type><name>firm_id</name></var>
<var><type>int</type><name>stock</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>
<!-- Message of stock and price from firm to mall  -->
<message>
<name>firm_stock_price</name>
<note>This message lets the people know how much stock the firm
they are buying from has left.</note>
<var><type>int</type><name>firm_id</name></var>
<var><type>int</type><name>stock</name></var>
<var><type>double</type><name>price</name></var>
<var><type>int</type><name>mall_id</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>
</messages>
<!--**** End of Messages ******-->

</xmachine_agent_model>
```

C Labour Market Model

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xmachine_agent_model>
<name>Labour Market</name>
<author>Eurace</author>
<date>290507</date>

<!--****** Environment values and functions *******-->
<environment>
<functions>
<file>Household_functions.c</file>
<file>Firm_functions.c</file>
<file>Eurostat_functions.c</file>
<file>Market_Research_functions.c</file>
<file>my_library_functions.c</file>
</functions>
<datatype>
<name>employee</name>
<desc>Used to hold employee information in firms</desc>
<var><type>int</type><name>id</name></var>
<var><type>int</type><name>wage</name></var>
</datatype>
<datatype>
<name>stock</name>
<desc>Used by households to hold stock information</desc>
<var><type>int</type><name>firm_id</name></var>
<var><type>int</type><name>price</name></var>
</datatype>
<datatype>
<name>vacancy</name>
<desc>Used by households to hold vacancy information</desc>
<var><type>int</type><name>firm_id</name></var>
<var><type>int</type><name>wage</name></var>
</datatype>
<datatype>
<name>job_application</name>
<desc>Used by firms to hold job applications</desc>
<var><type>int</type><name>worker_id</name></var>
<var><type>int</type><name>wage</name></var>
</datatype>
<datatype>
<name>job_offer</name>
<desc>Used by households and firms to hold job offers</desc>
<var><type>int</type><name>id</name></var>
<var><type>int</type><name>wage</name></var>
</datatype>
</environment>

<!--****** X-machine Agent - Firm **************-->
<xmachine>
<name>Firm</name>
<!--             Variables               -->
```

```
<!-- All variables used by Firm are declared here to
     allocate them in memory -->
<memory>
<var><type>int</type><name>id</name></var>
<var><type>employee_array</type><name>employees</name></var>
<var><type>int</type><name>wage_offer</name></var>
<var><type>int</type><name>technology</name></var>
<var><type>int</type><name>no_employees</name></var>
<var><type>int</type><name>vacancies</name></var>
<var><type>int</type><name>day_of_month_to_act</name></var>
<var><type>double</type><name>posx</name></var>
<var><type>double</type><name>posy</name></var>
</memory>
<!--     Defining functions     -->
<functions>

<function>
<name>Firm_read_job_applications_send_offer_or_rejection</name>
<depends>
<name>Household_read_job_redundency_vacancies_send_applications</name>
<type>job_application</type></depends>
</function>

<function>
<name>Firm_read_job_responses_update_wage_offer</name>
<depends>
<name>Household_read_job_offers_send_response</name>
<type>job_acceptance</type></depends>
</function>

<function><name>Firm_request_market_data_hyp</name>
</function>

<function><name>Firm_calc_redundencies_vacancies_and_send</name>
<depends>
<name>Market_Research_read_request_for_market_data_hyp_send</name>
<type>market_data</type></depends>
<depends>
<name>Eurostat_read_send_high_wage</name><type>high_wage</type>
</depends>
</function>

<function><name>Firm_request_high_wage</name>
</function>

</functions>
</xmachine>
<!--********* End of Agent - Firm ************************-->

<!--****** X-machine Agent - Household ******************-->
<xmachine>
<name>Household</name>
<!--      Variables for the Household        -->
<memory>
```

```
<var><type>int</type><name>id</name></var>
<var><type>int</type><name>wage</name></var>
<var><type>int</type><name>wage_reservation</name></var>
<var><type>int</type><name>skills</name></var>
<var><type>int</type><name>employee_firm_id</name></var>
<var><type>double</type><name>posx</name></var>
<var><type>double</type><name>posy</name></var>
</memory>
<!--        Defining functions                -->
<functions>

<function>
<name>Household_read_job_redundency_vacancies_send_applications</name>
<depends>
<name>Firm_calc_redundencies_vacancies_and_send</name>
<type>vacancies</type></depends>
<depends>
<name>Firm_calc_redundencies_vacancies_and_send</name>
<type>redundency</type></depends>
</function>

<function>
<name>Household_read_job_offers_send_response</name>
<depends>
<name>Firm_read_job_applications_send_offer_or_rejection</name>
<type>job_offer</type></depends>
</function>

<function>
<name>Household_read_application_rejection_update_wage_reservation</name>
<depends>
<name>Firm_read_job_applications_send_offer_or_rejection</name>
<type>job_rejection</type></depends>
<depends>
<name>Household_read_job_offers_send_response</name>
<type>internal</type></depends>
</function>

</functions>
</xmachine>
<!--**** End of Agent - Household ***************************-->

<xmachine>
<name>Market_Research</name>
<!--      Variables for the Market_Research        -->
<memory>
<var><type>int</type><name>id</name></var>
<var><type>double</type><name>posx</name></var>
<var><type>double</type><name>posy</name></var>
</memory>
<!--        Defining functions                -->
<functions>

<function>
```

```
<name>Market_Research_read_request_for_market_data_hyp_send</name>
<depends><name>Firm_request_market_data_hyp</name>
<type>market_data_request</type></depends>
</function>

</functions>
</xmachine>

<xmachine>
<name>Eurostat</name>
<memory>
<var><type>int</type><name>id</name></var>
<var><type>double</type><name>posx</name></var>
<var><type>double</type><name>posy</name></var>
</memory>
<!--        Defining functions                -->
<functions>

<function><name>Eurostat_read_send_high_wage</name>
<depends><name>Firm_request_high_wage</name>
<type>high_wage_request</type></depends>
</function>

</functions>
</xmachine>

<!--* Messages being posted by the agents to communicate *-->
<messages>

<message>
<name>high_wage_request</name>
<var><type>int</type><name>firm_id</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>

<message>
<name>high_wage</name>
<var><type>int</type><name>firm_id</name></var>
<var><type>int</type><name>high_wage</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>

<message>
<name>market_data_request</name>
<var><type>int</type><name>firm_id</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
```

```
<var><type>double</type><name>z</name></var>
</message>

<message>
<name>market_data</name>
<var><type>int</type><name>firm_id</name></var>
<var><type>int</type><name>market_data</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>

<message>
<name>vacancies</name>
<var><type>int</type><name>firm_id</name></var>
<var><type>int</type><name>firm_vacancies</name></var>
<var><type>int</type><name>firm_wage</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>

<message>
<name>job_application</name>
<var><type>int</type><name>firm_id</name></var>
<var><type>int</type><name>worker_id</name></var>
<var><type>int</type><name>wage</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>

<message>
<name>job_offer</name>
<var><type>int</type><name>worker_id</name></var>
<var><type>int</type><name>firm_id</name></var>
<var><type>int</type><name>wage</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>

<message>
<name>job_acceptance</name>
<var><type>int</type><name>firm_id</name></var>
<var><type>int</type><name>worker_id</name></var>
<var><type>int</type><name>wage</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
```

```
<var><type>double</type><name>z</name></var>
</message>

<message>
<name>job_rejection</name>
<var><type>int</type><name>worker_id</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>

<message>
<name>redundency</name>
<var><type>int</type><name>worker_id</name></var>
<var><type>double</type><name>range</name></var>
<var><type>double</type><name>x</name></var>
<var><type>double</type><name>y</name></var>
<var><type>double</type><name>z</name></var>
</message>

</messages>
<!--**** End of Messages *****-->

</xmachine_agent_model>
```

References

[1] T Balanescu, AJ Cowling, M Georgescu, M Holcombe, and C Vertan. Communicating stream X-machines are no more than X-machines. *Journal of Universal Computer Science*, 5(9):494–507, September 1999.

[2] J Barnard, J Whitworth, and M Woodward. Communicating x-machines. *Information and Software Technology*, 38(6):401–407, June 1996.

[3] B. Bauer, J.P. Muller, and J. Odell. Agent uml: a formalism for specifying multiagent software systems. *International Journal on Software Engineering and Knowledge Engineering (IJSEKE)*, 1(2), 2001.

[4] B. Bauer, J. Odell, and H. Parunak. Extending uml for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, *Proceedings of the Agent-Oriented Information Systems Workshop (AOIS), Austin*, pages 3 – 17, 2000.

[5] M. Catalano, F. Clementi, Domenico Delli Gatti, C. Di Guilmi, Edoardo Gaffeo, Mauro Gallegati, Gianfranco Giulioni, M. Napoletano, Antonio Palestrini, and A. Russo. The C@S project. EURACE Working Paper, September 22 2006.

[6] Simon Coakley. *Formal Software Architecture for Agent-Based Modelling in Biology*. PhD thesis, Department of Computer Science, University of Sheffield, Sheffield, UK, 2007.

[7] Samuel Eilenberg. Automata, languages and machines. Vol. A. Academic Press, London, 1974.

[8] Mike Holcombe. Towards a formal description of intracellular biochemical organisation. Technical Report CS-86-1, Dept of Computer Science, University of Sheffield, Sheffield, UK, 1986.

[9] Bernardo A. Huberman and Natalie S. Glance. Evolutionary games and computer simulations. *Proceedings of the National Academy of Sciences*, 90:7716 – 7718, August 1993.

[10] M. Huget. Agent uml class diagrams revisited. In J. Muller B. Rumpe B. Bauer, K. Fischer, editor, *Proceedings of Agent Technology and Software Engineering (AgeS), Erfurt, Germany*, 2002.

[11] Adrian Jackson. Single sided communication on hpcx. Technical Report HPCxTR0305, University of Edinburgh, October 2003.

[12] Gilbert Laycock. *The Theory and Practice of Specification Based Software Testing*. PhD thesis, Dept of Computer Science, University of Sheffield, Sheffield, UK, 1993.

[13] R.J. Pryor, D. Marozas, M. Allen, O. Paananen, K. Hiebert-Dodd, and R.K. Reinert. Modeling requirements for simulating the effects of extreme acts of terrorism: A white paper. Report SAND98-2289, SANDIA National Laboratories, 1998.

[14] Leigh Tesfatsion. Agent based computational economics, July 2007. <Online: `http://www.econ.iastate.edu/tesfatsi/ace.htm`>.

[15] Bielefeld University. Capital, consumption goods, and labour markets in eurace. EURACE Working paper WP5.1, April 2006.

[16] Sander van der Hoog and Christophe Deissenberg. Modelling requirements for EURACE. EURACE Working paper WP2.1, January 2007.

[17] Sander van der Hoog and Christophe Deissenberg. Modelling specifications for EURACE. EURACE Working paper WP2.2, January 2007.

[18] Boris Vaysburg, Luay H. Tahat, and Bogdan Korel. Dependence analysis in reduction of requirement based test suites. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 107–111, New York, NY, USA, 2002. ACM Press.

[19] G. Weisbuch, A. Kirman, and A. Herreiner. Market organization and trading relationships. *The Economic Journal*, 110:411 – 436, 2000.

Glossary

Eurostat : The statistical arm of the European Commission.

HPC : High Performance Computer – parallel supercomputer or computer cluster.

Node : Any single computer connected to a network. Supercomputer clusters are many up of many nodes.

NUTS-2 : Nomenclature of Territorial Units for Statistics – Used by Eurostat for E.C. regional statistics, level 2 being the region level.

UML : Unified Modelling Language – a standard notation and modelling technique for modelling software systems.

XML : Extensible Markup Language – a simple and very flexible text format designed for information exchange that encodes data with meaningful structure and semantics.