Report On C@S Project Model

Simon Coakley and Mariam Kiran

Department of Computer Science, Sheffield University, UK

October 13, 2006

Abstract

This report summarises the creation of a economics agent-based model, using the C@S project model, using the X-agents architecture.

1 Introduction

This is the first effort to create an economic agent model in the X-agents architecture. It uses the first paper circulated on the C@S project. Firms and workers/consumers have been implemented, with markets for jobs and goods. All the rules of the model are defined as part of each agent, with the communication between each agent is achieved with predefined message types.

2 X-Agents Framework

The framework used is called X-agents, and is the resultant work of Coakley's Ph.D. The driving force behind the work was to create a generic agent-based framework that was inherently parallel, testable, and made use of some formalisation to achieve this aim.

The starting point was the common use of grid based cellular automata, for example the game of life, and to make this more general. Firstly grid cells in cellular automata were thought of as finite state machines, with inputs being the states of its neighbours, and rules encoded into the state transitions. To make this model more general, the idea of an extended finite state machine was used, in this case defined using the communicating X-machine model. This model adds memory to the machine, as well as the ability to send and receive messages. The position of each machine, or agent, can now be defined as values in memory, for example position on the x and y axes. Unfortunately communication is no longer static, as there is no grid and machines can change their position. Communication is now achieved via sending and reading messages to and from message lists, which can be

defined as part of the model. A message list is created for each message type and holds every message of that type that has been sent during an iteration, with new messages added to the top of the list.

A model is defined in the framework, at the lowest level, as an XML document. This uses tags to markup the components of the model so that it can be easily parsed into source code that can be compiled into a runnable simulation. Each agent in the model must define its memory and functions. For the moment, design changes are foreseen, each agent must have the same amount of functions. This is so that the agents can be keep synchronised easily. Messages that agents use to communicate must also be defined as port of the model.

The whole point of keeping agents autonomous, i.e. only agents can access and change their own memory, and communication is only achieved via messages is that agents and models become a lot easier to test, and the work of parallelisation is done for you, albeit handling the message lists on different nodes on a parallel machine.

Please take into account that although the framework has been successfully used for biological modelling, there are still many design decisions to make with regard to economic models and the general design of the architecture.

3 Implementation

The following section describes the attempt to implement the C@S project model, using firms and workers/consumers. Firstly agent memory is defined, then the messages used, then the agent functions, including their order, with reference to the C@S paper sequence of events.

3.1 Agent's Memory

Two agents called Firm, see Table 1, and Person, see Table 2, are defined. These variables can be accessed via *set* and *get* methods or, for the moment, via the C dereferencing operator `->` from the structure *xmemory*, in the agent functions.

3.2 Messages

See Table 3 for the messages used in the communication between agents. When messages have been defined, agent functions can interact with the associated message list via the framework inbuilt functions:

- `add_`*messagename*`_message{`*messagevariables*`}` – send a message of the specified type

Name	Type	Description
id	int	Identification number
value	double	Equity
a	double	Financial viability
productivity	double	Productivity
profits	double	Profits
f	double	Fraction of R&D from profits
production	double	Expected production
goodsproduced	int	Amount of goods actually produced
stock	int	Amount of goods not yet sold
labour	int	Amount of workers needed
numberofworkers	int	Amount of workers hired
price	double	Final price of goods
oldprice	double	Last iteration goods price
sprice	double	Satisficing price
lprice	double	Lowest price
oldworkerid[100]	int	List to hold job applications from previously employed workers
oldworkerwage[100]	int	and their new wages
newworkerid[100]	int	List to hold job applications from new workers
newworkerwage[100]	int	and their new wages
hiredworkerid[100]	int	List to hold hired workers
hiredworkerwage[100]	int	and their wages
posx	double	Legacy value, possible future use in geographical positioning
posy	double	Ditto
iradius	double	Legacy value, interaction radius when using positioning

Table 1: Memory of Firm agent

Name	Type	Description
id	int	Identification number
savings	double	Income of the worker (from previous iterations)
wage	double	Wage the worker is earning
firmid	int	Firm the worker was last employed by
posx	double	Legacy value, possible future use in geographical positioning
posy	double	Ditto
iradius	double	Legacy value, interaction radius when using positioning

Table 2: Memory of Person agent

- `get_first_`*messagename*`_message()` – returns the first message in the message list of the specified type
- `get_next_`*messagename*`_message(`*currentmessage*`)` – return the next message in the list

Name: priceinflation Description: holds the price inflation sent by each firm		
priceinflation	int	The price inflation value

Name: application Description: a job application message from a worker to a firm		
person_id	int	The id of the worker
person_wage	double	The wage wanted by the worker
last_firm_id	int	The last firm the worker was employed by
to_firm_id	int	The firm applying to

Name: employed Description: the message from firms to workers saying they are hired		
person_id	int	The worker hired
firm_id	int	The hiring firm

Name: goods_price Description: holds the price of the goods from each firm		
firm_id	int	The firm
goods_price	double	The price of goods

Name: firm_stock Description: the amount of stock a firm has		
firm_id	int	The firm
stock	int	The amount of stock left

Table 3: Messages used for communication

3.3 Functions

The sequence of events have been mapped to functions in the model, see Table 4. Please refer to the source code of the functions in the model file for full explanations of the implementation.

3.4 Starting Values

Now a model has been defined, a simulation can be started be defining agents that are going to take part in the simulation. This is achieved by defining

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 4: Sequence of events

the value of each agents variables in a start states file, also marked up in XML. When a simulation runs, for each iteration, a results file is produced of the same structure as a start states file. Once a simulation can been completed these results files can be parsed and the changes in variables can be viewed as graphs or images.