An Agent-Based software platform for modelling systems biology

Salem F. Adra
Simon Coakley
Mariam Kiran
Phil McMinn

Advisors: Prof. Rod Smallwood and Prof. Mike Holcombe

2008

Contents

List of Figures

List of Tables

Executive Summary

The Epitheliome project[1] aims to use agent-based modelling to develop a computational model that is able to predict the emergent behaviour of cells in epithelial tissues. Agent-based modelling provides more innovative approaches to facilitating research into the unresolved issues of complex systems. The huge amount of data provided by the sequencing of the human genome and the variant time scale (milliseconds to months) of processes taking place within the tissue are among the complexities involved in understanding biological processes. Computational models are believed to be of great utility managing such complexities and enhancing our understanding of biological systems. This document gives insights into the modelling framework, FLAME, and how it can be applied to biological modelling.

An agent-based modelling framework, FLAME [5], developed at the University of Sheffield, has been successfully used to model biological systems and has already uncovered some 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 are described in the report:

- The framework uses XMML, X-machine markup modelling language, to define agents and the communications between them.
- Various features are provided by XMML and the framework that allow modellers to design their own models and simulate them over time.
- A few examples of its application to biological models have proven its success and are presented here.
- The project team work closely with biologists from the universities of Sheffield and York (Jack Birch Unit for Molecular Carcinogenesis) to understand and interpret cell behaviour, and use this information to produce accurate models.
- The University of Sheffield also works with the Rutherford Appleton Laboratories to produce efficient models for deployment on parallel platforms.

Various examples of biological models have been produced highlighting the success of the framework and XMML. These include the keratinocyte (skin) [15] and the Urothelial Cell (bladder) [18] models. Results of the keratinocyte cell model are illustrated in this document.

This document presents the framework, and shows how the XMML schema can be used to produce various models of the epithelial cells.

[1]The main project is supported by the EPSRC (Engineering and Physical Sciences Research Council) by a project grant of £2m.

1 Introduction

This document contains details of agent based simulations of biological systems. The document also presents a definition of the XMML modelling language and how models can be written using it.

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;
- **Biological Models: The Keratinocyte Model** - Presents the Keratinocyte, or skin cell, model;
- **Framework Implementation** - Contains implementation details of the framework;
- **Model Creation** - Contains details about how to construct models;
- **Links to External Solvers: COPASI Example** - Contains details about how to construct models which uses COPASI
- **Appendix A – XMML Schema**, which formally defines the XMML language;
- **Appendix B – Keratinocyte Model XMML**, which formally defines the Keratinocyte model;

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 [16] describes, 'refers broadly to a bundle of data and behavioural methods representing an entity constituting part of a computationally constructed world'. In computational systems biology, the definition of an agent can also vary from representing a group of agents like a colony composed of many cells or an individual cell like a stem cell or a dead cell.

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 [6] 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 [13], 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.

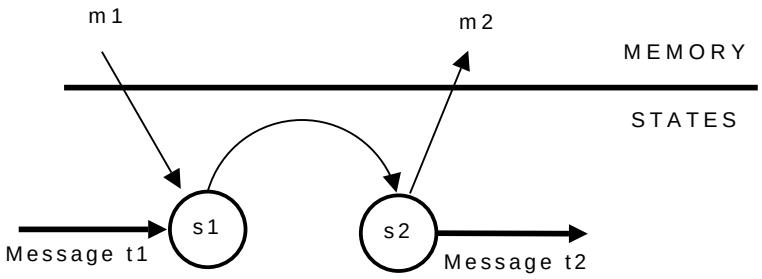

Figure 1: Transition function

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 an 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 8 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:

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

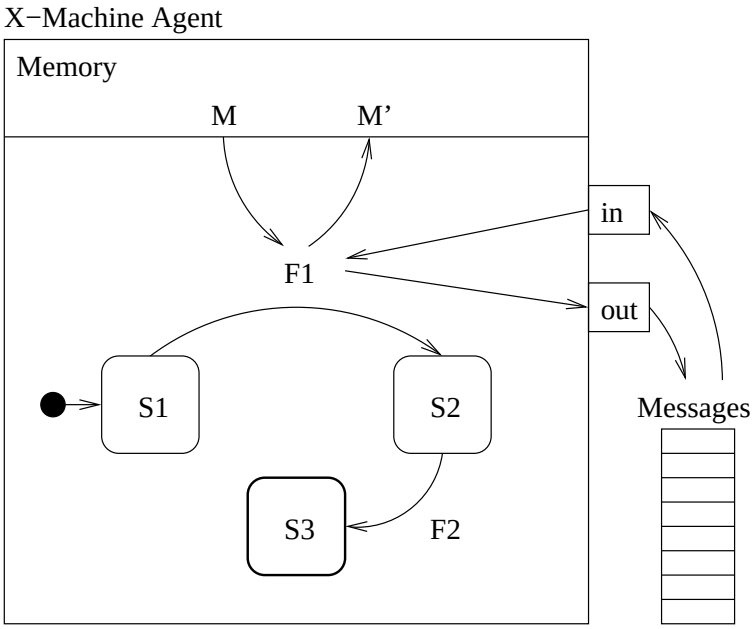

Figure 2: X-machine agent

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

This chapter describes the implementation issues surrounding the requirements/ specifications needed to build models of systems biology. These issues include formally specifying agents, transforming the model's 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 should highlight the following issues for building high-fidelity, high-resolution agent-based models as described by Pryor et al. 1998 [14]:

- 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 proposed as a way to design agent-based models [4, 3, 11, 20] 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 [2]. 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

[2] This is similar to branch traversal testing.

3.3 Biology Example: Keratinocyte colony formation

Normal human keratinocyte (NHK) is a cell type that constitutes over 80% of the outermost layer of the skin or the epidermis [7]. The epidermis is a fast renewing tissue which forms a protective barrier between our internal organs and the outside world. Understanding how cells proliferate and self-organise into layers of skin tissue is a very important research topic. Such understanding promotes the development of methods to artificially produce reconstructed human skin for patients with heavy skin loss for example through chronic burns, wounds or skin disease. As part of the Epitheliome project at the University of Sheffield, Sun and McMinn used the agent-based modelling framework, FLAME, to develop an in-virtuo model of the behaviour of skin cells. The interaction of the software-agents in the in-virtuo model described the NHK macroscopic morphogenesis in-vitro [15, 19]. The initial set of rules implemented to govern the cellular (agents) behaviour in-virtuo were based on well established areas of epidermal/dermal biology literature. As the main purpose of the model was to enhance the understanding of cell colony formation, many complex elements of the natural biology have been abstracted away. In the agent based model of the keratinocyte colony formation, each cell was represented by an individual agent. The signalling process between cells is simulated in the model in terms of agents communicating with each other by reading and writing to message lists. The algorithm for the keratinocyte colony formation can be summarised as follows:

1. Cells communicate with each other by exchanging information about their types and locations
2. Cells act accordingly and go into or continue a cell cycle [18] which includes several checkpoints
3. Cells divide depending on certain conditions (location, calcium concentration in the environment and number of contacts with other cells)
4. Cells differentiate depending on certain conditions (type, location and calcium concentration in the environment)
5. Cells migrate depending on certain conditions (type, location and calcium concentration in the environment)
6. A physical model controls the physical distribution of cells, making sure that any two cells will not physically overlap

The sequence of operations described are meant to cover one single time-step in the simulation. Every single time-step of the in-virtuo model represents a time-step of 30 minutes in reality. Following the method for creating an X-machine model, the cell agent system functions can start to be identified:

- Signal (used to communicate cell location)

- Cycle (cell cycle rules)
- Divide (Cell division or Mitosis)
- Differentiate (diff)
- Migrate
- Resolve location (used to ensure realistic cell locations)

The keratinocyte colony formation model, implemented by Sun *et al* [15] and used in here to illustrate the use of FLAME, adopted a simple approach to cell cycling alongside a simple physical model to resolve cell locations. Moreover, the in-virtuo model did not involve any complex cell signalling mechanisms and modelled each cell as a simple sphere, with constant size and shape, on a virtual culture dish.

At every iteration of the in-virtuo model simulation, each cell (agent) broadcasts its location to the message lists for all other cells to read. In the keratinocyte colony formation model, there are four types of cells considered: Stem cells (S), Transit Amplifying (TA) cells, Committed cells (comm) and Corneocytes (corn). Depending on its type and position in the cell cycle, each cell performs specific programmed rules of its cell cycle. Stem cells are usually found in the centre of the colony. They are fairly static and proliferate provided that there is space to do so. Similarly, TA cells also proliferate as long as there is sufficient space. As a result, each stem and TA cell follows a cell cycle whereby they divide after a pre-determined period of time. A stem cell division (mitosis) produces two daughter stem cells, and a TA cell division produces two daughter TA cells. If a cell is contact-inhibited, that is, there is no sufficient space around to divide, it goes into a special dormant state of the cell cycle referred to as G_0.

After executing the cell cycle rules, each cell then uses the differentiation rules adopted in the model to decide whether to change to another cell type (a process called differentiation). A stem cell can differentiate into a TA cell, a TA cell can differentiate into a committed cell. Initially, stem cells divide and cluster (Figure 3). When the stem cell cluster reaches a certain size, the cells located on the cluster edge, start to differentiate into TA cells. In practice this rule is coded by each stem cell scanning its vicinity through interrogation of the message lists (Figure 4). If a cell (agent) cannot find a certain number of fellow stem cells within a certain range d1, the cell differentiates into a TA cell.

Moreover, following the differentiation of stem cells into TA cells, when the TA cells become a distance d2 away from the stem cell epicentre of the colony, they differentiate into committed cells (Figure 5). Stem and TA cells that have been in G_0 for a certain period of time also differentiate into committed cells (Figure 6). Finally a committed cell will loose its nucleus and differentiate into a corneocyte after a further period of time.

Following the differentiation rules, each cell, depending on its type and location, then execute the migration rule. Unlike stem cells, TA cells can migrate around the culture plate, depending on the ambient levels of calcium. Lastly, when a stem, TA or committed cell dies (for example, due

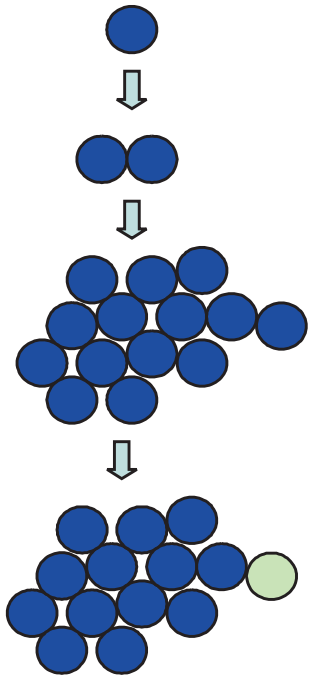

Figure 3: Differentiation from stem cells to TA cells. Stem cells are blue, TA cells are green.

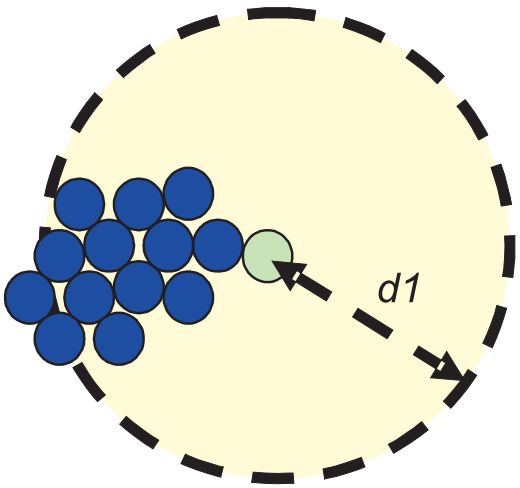

Figure 4: Mechanism for stem to TA cell differentiation

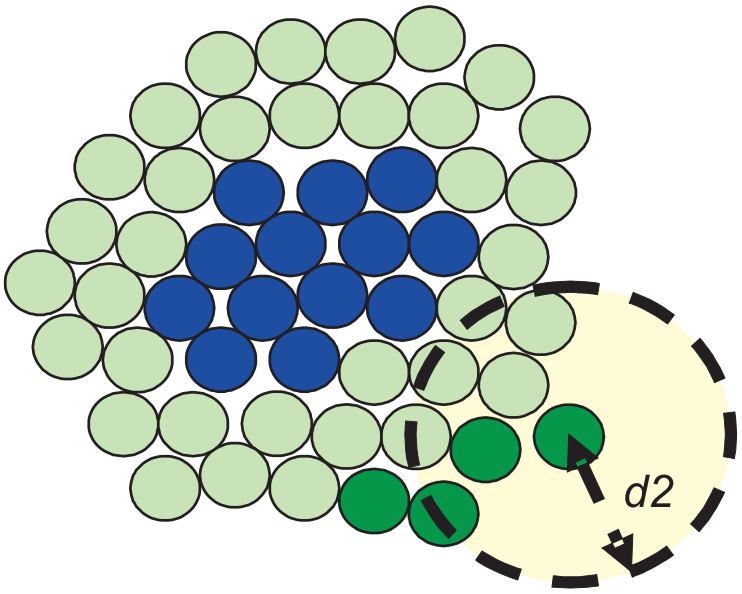

Figure 5: TA cells now dominate the edges of the colony. As the colony gets bigger TA cells far away from the stem cell epicentre differentiate into committed (dark green) cells.

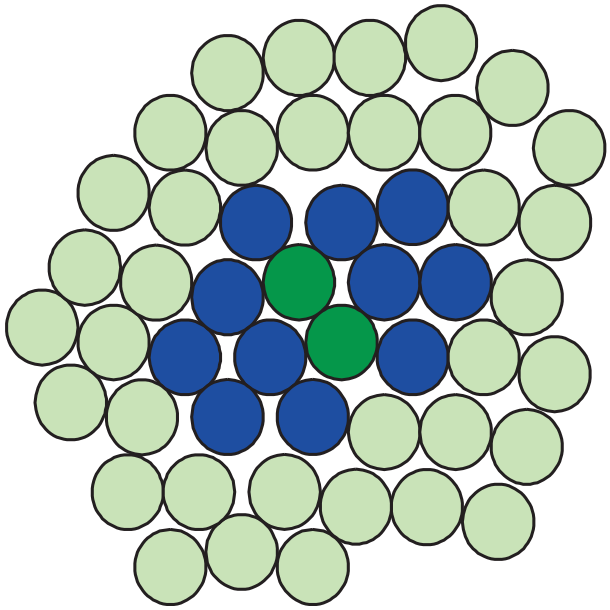

Figure 6: Stem and TA cells that have been contact-inhibited for a certain period of time differentiate into committed cells.

to an increase of calcium level in the environment), the cell differentiates into a corneocyte cell. Corneocytes are dead cells that do not proliferate, differentiate or migrate, and are found on the top layers of the epidermis.

At the end of every iteration (time-step) of the simulation, the model's physical rules are executed to ensure that the cell locations are realistic. In this example, the physical rules are executed by deploying an external physical solver agent. The physical solver simply applies a force to overlapping cells to separate them. Skin cells self-organise into layers through stratification. In the in-virtuo model, daughter cells are pushed up a layer when there is no lateral space (Figure 7). The physical model is also built so that different cells exert different forces. For example, while TA cells possess higher motility and can migrate, stem cells, committed cells and corneocytes bond strongly to themselves and the culture plate, which is why they remain static.

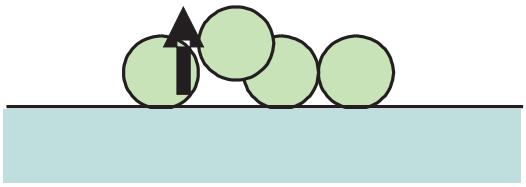

Figure 7: Stratification

A pseudocode description of the in-virtuo model of the keratinocyte colony formation can be summarised as follows [15]:

```
For each time-step
    For each agent
        Read state and position of neighbouring agents from message list
        Update state and position as determined by internal rules and
        external signals
        Write new state and position to message list
    End
End
```

Following the identification of the cell agent functions, 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 (the start and finish state can be the same state), and are shown in Table 1. In the keratinocyte model, a cell has many start states because a simulation can start with many cells of different types (Stem, TA, Corneocyte and Committed). A state transition diagram of the full in-virtuo model of keratinocyte colony formation is presented in Figure 8. In addition to 'signal', 'cycle', 'migrate', 'divide' and 'differentiate' functions, a cell can also 'not_divide ' and 'not_diff '.

Each cell (agent) stores its spatial co-ordinates in its local memory, and behaves according to programmed rules for movement, cell division and differentiation.// The cell memory is defined thus,

- Cell memory:
  - double x, y, z // location in 3D
  - contacts // used to count cell contacts

and the cell messages are defined as,

- Messages:
  - struct location { int cell_type, double x,y,z}.

Start State	Function	End State
Stem	signal, cycle, divide, or differentiate	Stem, Transit Amplifying or Corneocyte
Transit Amplifying (TA)	signal, cycle, divide, differentiate or migrate	Transit Amplifying, Committed or Corneocyte
Committed	signal, cycle or differentiate	Committed or Corneocyte
Corneocyte	signal or cycle	Corneocyte

Table 1: Cell states in the Keratinocyte colony formation model

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, where try_cycle, try_divide, migrate, Physical_solver, and resolve_locations are defined in Tables 4-8. The keratinocyte colony formation defined in XMML is presented in Appendix B.

Input	Function	Output
	Signal	location out
location in	Cycle	
	Divide	
location in	Diff	
	Migrate	
	Resolve_location	

Table 2: Cell input and output messages

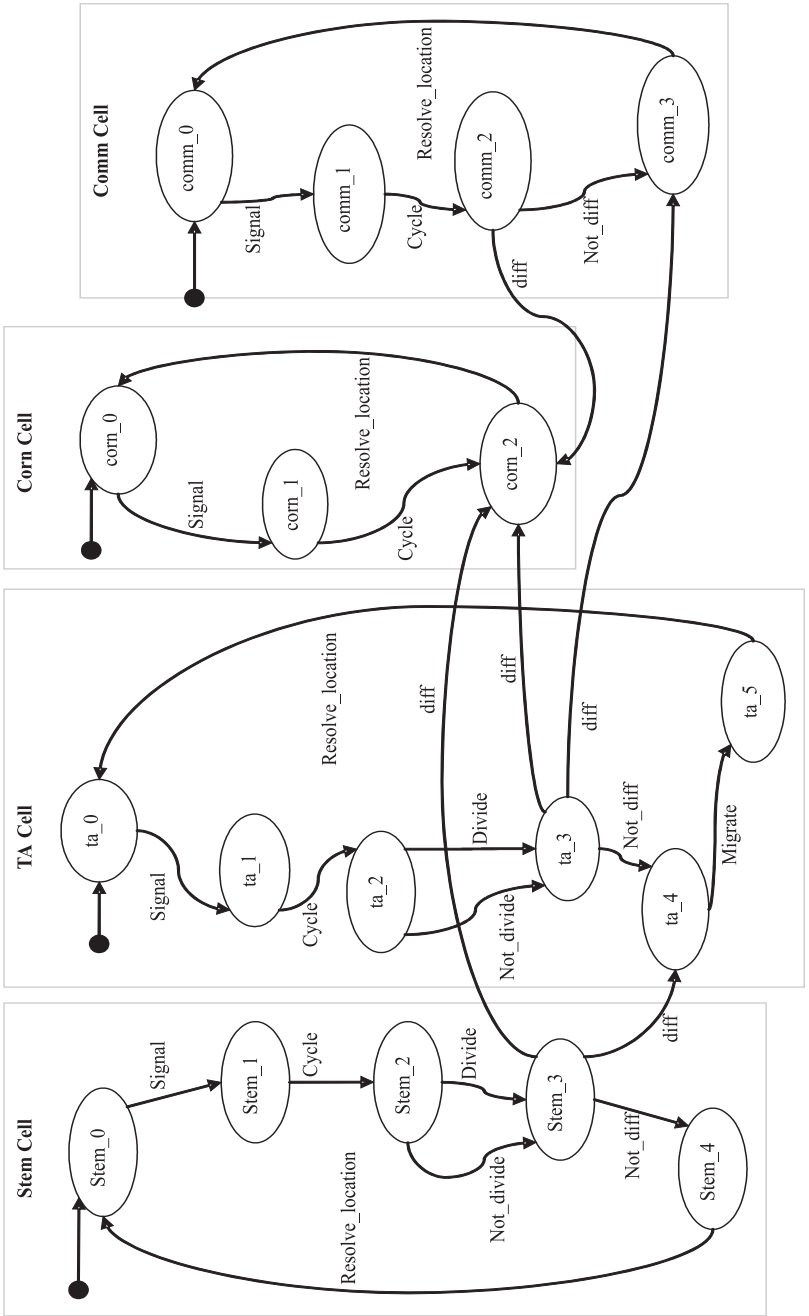

Figure 8: States transition diagram

Start State	M_{pre}	Input	Function	End State	M_{post}	Output
stem_0			signal	stem_1		location
ta_0			signal	ta_1		location
comm_0			signal	comm_1		location
corn_0			signal	corn_1		location
stem_1		location	try_cycle	stem_2		
ta_1		location	try_cycle	ta_2		
comm_1		location	try_cycle	comm_2		
corn_1		location	try_cycle	corn_2		
stem_2			try_divide	stem_3		
ta_2			try_divide	ta_3		
stem_3		number of stem neighbours > max stem colony size	diff	ta_4		
stem_3		$z > 0$	diff	ta_4		
stem_3		contact inhibited ticks > max to g0 contact inhibited ticks	diff	ta_4		
ta_3		nearest stem cell > same distance	diff	corn_2		
ta_3		contact inhibited ticks > max to g0 contact inhibited ticks	diff	comm_3		
ta_3		number of corn neighbours > some value	diff	corn_2		
comm_2		dead ticks > max dead ticks	diff	corn_2		
comm_2		number of corn neighbours > some value	diff	corn_2		
ta_3			not_diff	ta_4		
comm_2			not_diff	comm_3		
stem_3			not_diff	stem_4		
ta_4			migrate	ta_5		new_location
comm_3			not_migrate	comm_4		new_location
corn_2			not_migrate	corn_3		new_location
stem_4			not_migrate	stem_5		new_location
ta_5		res_location	resolved_location	ta_0		
comm_4		res_location	resolved_location	comm_0		
corn_3		res_location	resolved_location	corn_0		
stem_5		res_location	resolved_location	stem_0		

Table 3: Cell memory pre and post state transitions

Start State	M_{pre}	Input	Function	End State	M_{post}	Output
Start			init_counts	1	$contacts = 0;$ $num\_stem\_neighbours = 0;$ $num\_corn\_neighbours = 0;$	
1	$distance(x, y, z, location.x, .y, .z) <= force\_radius$ and location.cell_type == STEM	location	cell_contact_stem	1	$num\_stem\_neighbours++;$ $contacts++;$	
1	$distance(x, y, z, location.x, .y, .z) <= force\_radius$ and location.cell_type == CORN	location	cell_contact_corn	1	$num\_corn\_neighbours++;$ $contacts++;$	
1	$distance(x, y, z, location.x, .y, .z) <= force\_radius$ and location.cell_type != STEM and location.cell_type != CORN	location	cell_contact	1	$contacts++;$	
1	$distance(x, y, z, location.x, .y, .z) > force\_radius$	location	no_cell_contact	1	$contacts++;$	
1		$location == null$	count_contacts_finished	2		
2	$(calcium\_level == 0.1 and contacts <= 4)$ or $(calcium\_level == 1.3 and contacts <= 6)$		cycle	End	$cycle++;$ $contact\_inhibited\_ticks = 0;$	
2	$(calcium\_level == 0.1 and contacts > 4)$ or $(calcium\_level == 1.3 and contacts > 6)$		no_cycle	End	$contact\_inhibited\_ticks++;$	

Table 4: Try_Cycle

Start State	M_{pre}	Input	Function	End State	M_{post}	Output
1	Cycle $>$ 60		divide	End	$cycle = random(0, 15)$; add_cell(ta);	
1	Cycle <= 60		no_divide	End	$cycle = random(0, 15)$; add_cell(ta);	

Table 5: Try_divide

Start State	M_{pre}	Input	Function	End State	M_{post}	Output
Start			migrate	End	$x = x + mobility + cos(direction); y = y + mobility + sin(direction);$	

Table 6: Migrate

Start State	M_{pre}	Input	Function	End State	M_{post}	Output
Phy_sol_0			resolve_locations	Phy_sol_1		

Table 7: Physical solver

Start State	M_{pre}	Input	Function	End State	M_{post}	Output
Start			init_locations	1	locations.init();	
1		new_location	add_new_location	1	locations.add(location);	
1		new_location == null	solve_locations	2	call_external_solver();	
2	$locations.size > 0$		send_location	2	$location = locations[0]$;	res_location (location)
2	$locations.size == 0$		finish_sending_locations	End		

Table 8: resolve_locations

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. Alternatively, a memory value could include an internal clock ticker, as in the case of the in-virtuo model of keratinocyte colony formation, where the agent waits for a certain amount of clock ticks before transitioning into different states. In such scenarios, there should 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.

3.4 Unified Modelling Framework

By creating a unified modelling framework other people can use their expertise to create their own models. 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 the Rutherford Appleton Laboratory 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 9.

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.

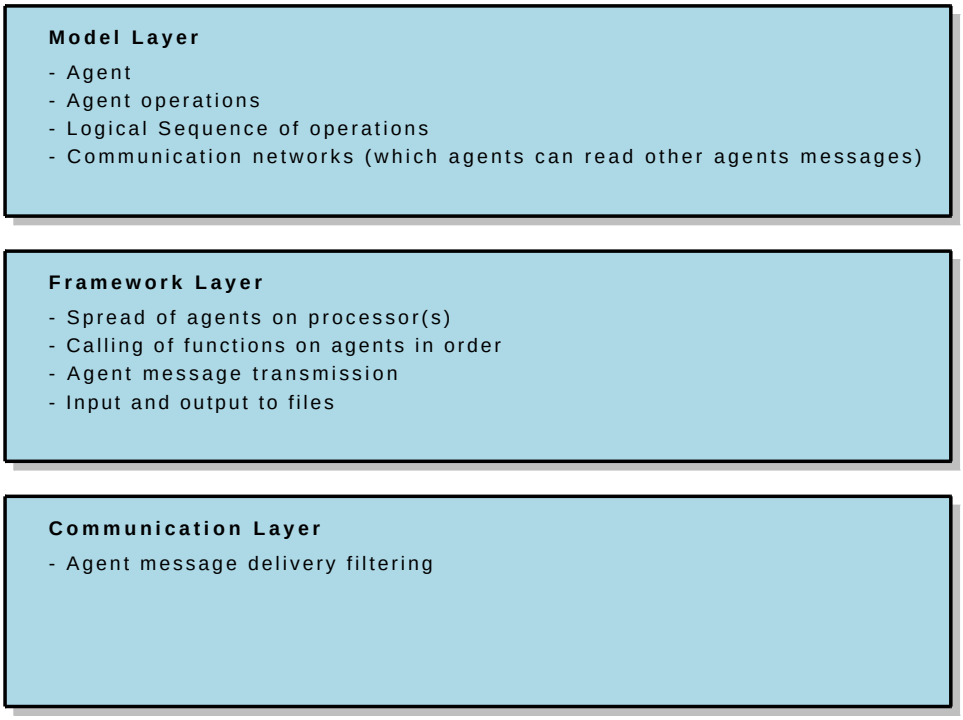

Figure 9: Layers of abstraction for the framework.

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 model 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 [10].

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:

- 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 Rutherford Appleton Laboratories 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 [12]. 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 [3]. 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 [17] from 2002 uses

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

this dependence analysis technique to aid automated test case generation, which could also aid testing of models in the framework.

Figure 10 shows a dependency graph for the in-virtuo model of keratinocyte colony formation. The dependency graph shows all the actions that can happen in one time-step (iteration), i.e. after an 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.

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 interaction).
- 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.

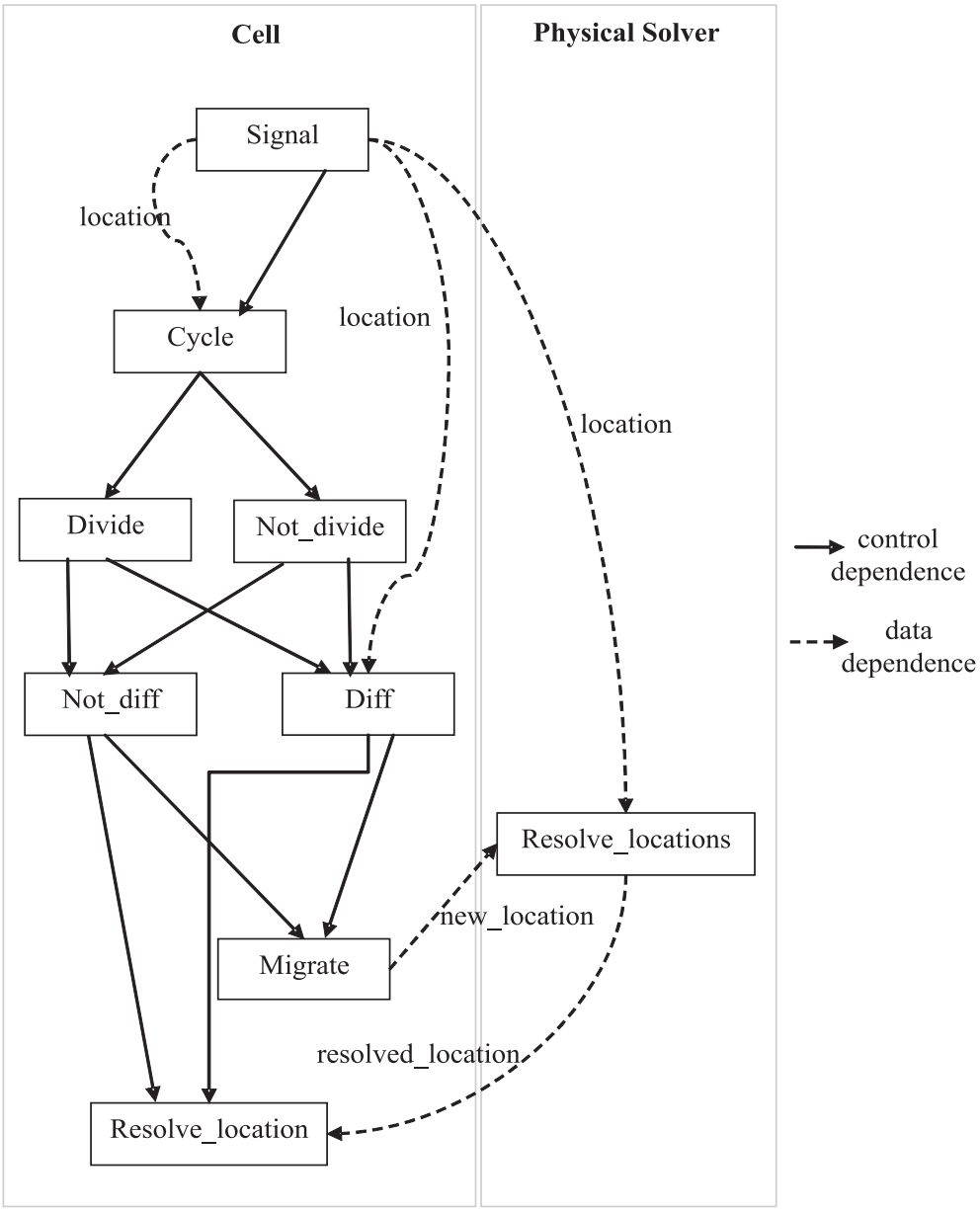

Figure 10: Dependence graph

- 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

Discussions and experiments with these and other file formats are currently being performed by Sheffield and Rutherford Appleton Laboratories.

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 11. 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.

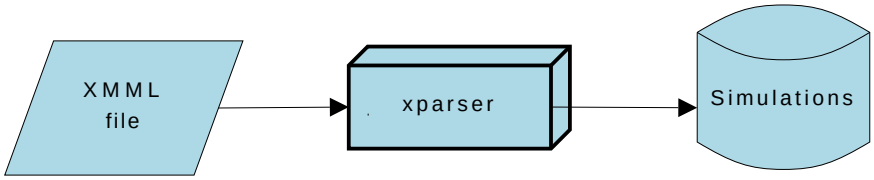

Figure 11: Xparser usage

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 12 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 12 into layers separated by synchronisation blocks in Figure 13.

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

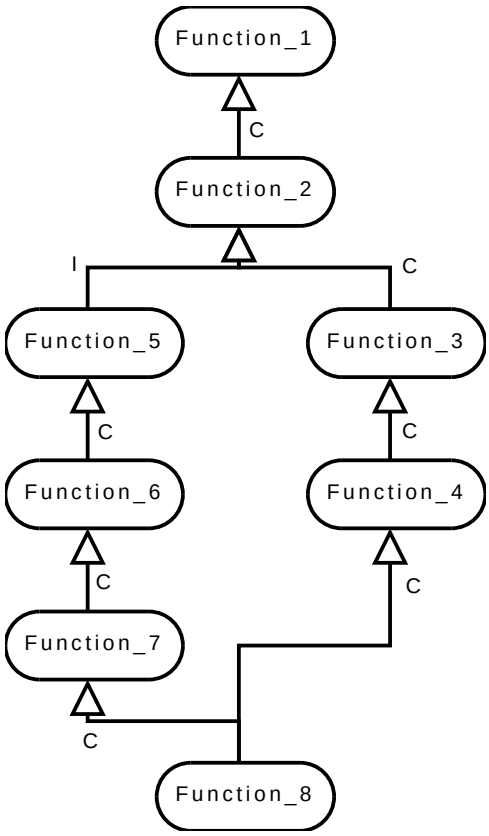

Figure 12: Communication dependencies between functions

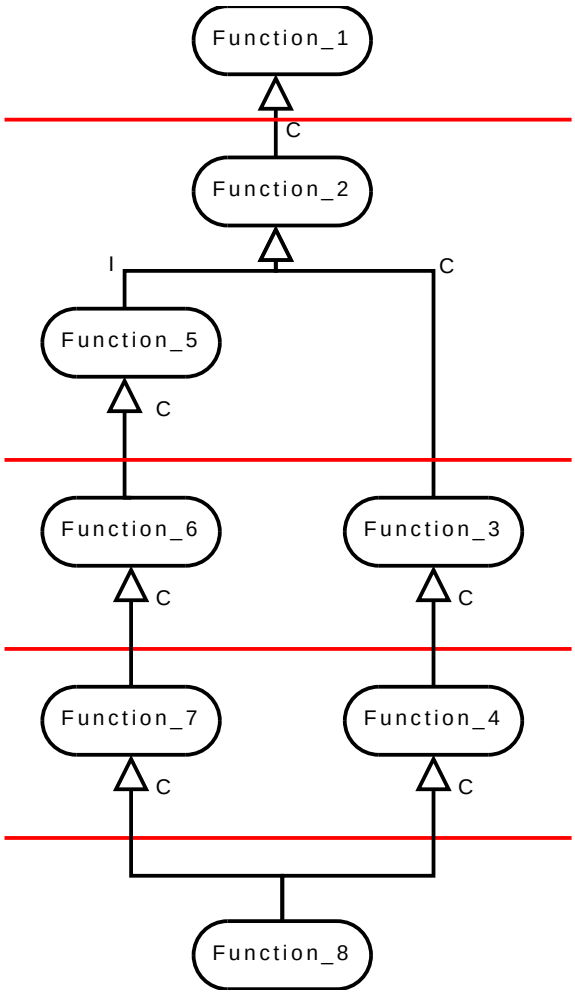

Figure 13: Syncing communication dependencies as synchronisation layers

the message range. After discussions with members of the Rutherford Appleton Laboratories about 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 [12], 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.

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 9.

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 9: C fundamental data types.

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 10 gives an example of a custom data type called *Chemical_Element* which holds an ‘Atomic_number ’ of type *int* and a ‘concentration’ of type *double*.

```
<datatype>
<name>Chemical_Element </name>
<desc>Used to hold a chemical element information</desc>
<var><type>int</type><name>atomic_number </name></var>
<var><type>double</type><name>concentration</name></var>
</datatype>
```

Table 10: Example of the Chemical_Element data type.

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 reactants, the definition would be (Table 11):

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

Table 11: 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 12):

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

Table 12: 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.

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 13 gives an example of a firm agent.

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

Table 13: Example of a Cell Agent.

```
<messages>
<!−−Message for neighbouring cells−−−−−>
<message>
<name>location</name>
<note>This message lets the cells know about the location of the cell
sending the message.</note>
<var><type>int</type><name>cell_id</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>
```

Table 14: Example of describing messages.

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 separately. An example has been shown in Table 14.

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 9).
- 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 Outputs Produced by the Xparser

The Xparser produces simulation source code files, a compilation script, and a documentation options file. Also produced are two graphs that show function dependencies (see Figure 14 for example) and function order with communication layers (see Figure 15 for an example).

5.6.1 Dotty graphs

The communication between agents takes place between agent functions. In other words, one agent function sends a message, and another agent function reads the message. The dotty graphs illustrate the order of agent functions (i.e. creating a dependency graph from the function dependencies). Figure 14 is an example of the output produced. When one function depends on a message sent from another function this is called a communication function dependency. On the other hand, when one function depends on the outcome of another functions within the same agent, this is called an internal function dependency.

5.6.2 SVG graphs

The svg files, produced after compilation with the xparser, allow a clear understanding on how the functions will be ordered during execution. Figure 15 depicts the output of the svg files. Red lines depict a synchronisation 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 synchronisation point.

5.6.3 Results and Conclusions

The keratinocyte colony formation model provides a test bed for:

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

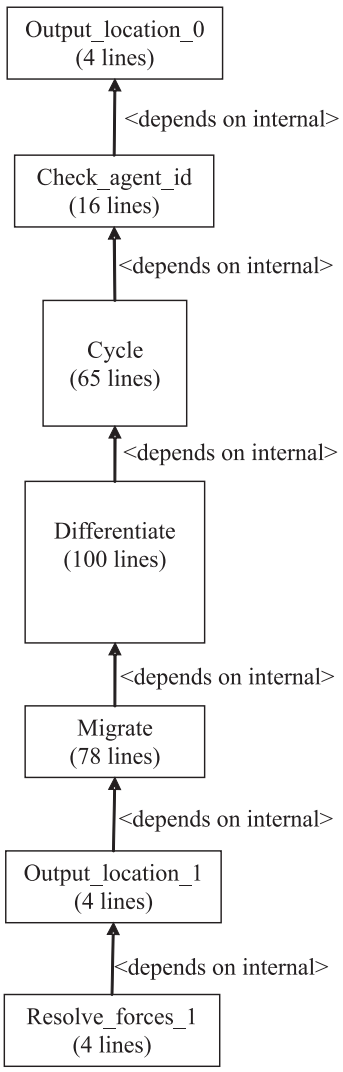

Figure 14: Function dependency graph of keratinocyte colony formation model

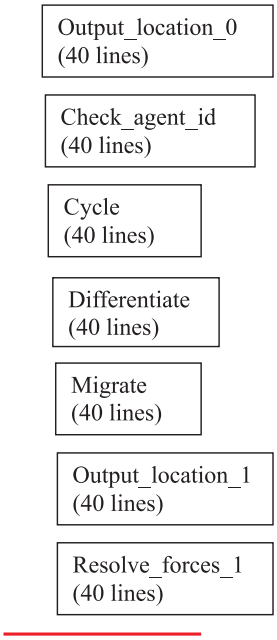

Figure 15: Communication synchronisation layers of keratinocyte model

Running models efficiently includes calculating when best to run functions and where to place communication synchronisation points between functions. Figure 15 lists the functions on rows with communication points as a red line(s). 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.

5.7 Links to External Solvers: COPASI Example

5.7.1 External Solvers Integration

The ability to link FLAME to external solvers is very beneficial for several reasons:

- It makes it possible to use existing resources and tools to simulate or solve certain model aspects such as differential equations
- It facilitates and accelerates the process of building models and allows researchers to focus on application specific modeling issues
- It helps bringing together people from different modeling backgrounds and encourages model reuse and communication

5.7.2 COPASI

As a starter, the COmplex PAthway SImulator software application (COPASI) [9] was chosen to be interfaced with FLAME. COPASI is a software application used for simulating and analyzing biochemical networks. COPASI is a free software for non-commercial use. Moreover, COPASI provides alot of useful functionalities such as stochastic and deterministic time course simulation, steady state analysis, metabolic control analysis and optimisation of arbitrary objective functions. COPASI includes a graphical user interface for building and simulating biochemical models and it also provides a command line version for batch processing. One of COPASI attractions is that it can import and export SBML (Systems Biology Markup Language). COPASI can also export the ODE system it automatically generates from the chemical reactions and the global quantities defined by the user to C source code.

5.7.3 Interfacing FLAME and COPASI

FLAME is interfaced with COPASI as an external solver. Some potential usage for such interfacing is the modeling (time course simulation) of intracellular biochemical reactions and signalling pathways (e.g. TGFβ signalling pathway) which can be used to regulate the behaviour of the cell agents.

When the use of COPASI is required, the modeler need to define within the $< environment >$ tag (in the xml file defining the model) the data structure **copasi_data** which should contain the name (string of characters) and the concentration (real number) of a metabolite. This is illustrated in Appendix C. The initialisation file provided by the user should then look something like the xml file presented in Appendix D.

Interfacing FLAME with COPASI is realised by providing a set of utility functions that constitutes a COPASI-Interfacing Toolbox.

The Toolbox Content:

- **CopasiModelTemplate.tmpl:**
  A basic and empty COPASI model (.cps) which is used to create a COPASI model based on the user defined data structure **copasi_data** (In the xmml model definition file). The COPASI model is then populated using the corresponding values (metabolites names and concentrations) defined in the initilisation file (e.g. Appendinx D).

- **CopasiFunctions.c:**
  Contains two Functions:
  initialise_Copasi_Model and **update_Copasi_Model**.
  The function **update_Copasi_Model** is called at every iteration of the model simulation to update the variables (metabolites names and concentrations) in the COPASI model based on the current values of an agent memory. The COPASI model is then executed (Time course simulation task), and the new metabolites' concentrations stored in an automatically defined and formated output report are used to modify the agents memory correspondingly

- **Initialise_copasi_model.c:**
  An executable file (once the codes generated by the parser are compiled) which should be called before the main executable (main.exe) which simulates the model for a certain user-defined number of iterations.
  Initialise_copasi_model.c calls the function **initialise_Copasi_Model** and uses the template **CopasiModelTemplate.tmpl** to create the COPASI model

In the file *Functions.c*, provided by the modeller to define the agents behaviour, the modeller also needs to define the following variables:

- A String defining the name of the COPASI model file:
  e.g. **copasiModel** = "TGFBeta.cps";

- A String defining the name of the COPASI model:
  e.g. **copasiModelName** = "TGFBeta";

- A String defining the name of the output report file:
  e.g. **reportFile** = "CopasiModel.txt";

- A String defining the name of the output report:
  e.g. **reportName** = "Flame_Report";

- A String defining the desired rate law interpration method:
  rateLawInterpretation = "stochastic" or "deterministic";

- A String defining the quantity unit:
  quantityUnit = "#"; The default value is "mmol"

- A String defining the time unit:
  timeUnit = "min" or "s" or "h";
- An Integer defining the duration of the time course simulation:
  e.g. **duration** = 30;
- An integer defining the number of intervals that divides the duration of the time course simulation:
  e.g. **nbofIntervals** = 100;

Unless loading an existing COPASI model with defined reactions, after calling the executable *Initialise_copasi_model.exe*, the user need to open the COPASI model (.cps file) defined by the string *copasiModel* using the COPASI GUI and set the reactions needed for the time course simulation of the defined metabolites. The main executable *main.exe* can then be called to simulate the model for a certain number of iterations. The results (memory value for the agents memory, including the COPASI metabolites) will then be written to xml files at every iteration. The results achieved at the 10^{th} iteration, for example, are shown in Appendix E.

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 Keratinocyte Colony Formation Model

```
<?xml version="1.0" ?>

<xmachine_agent_model>
<name>Keratinocyte colony model</name>
<author>Phil McMinn</author>

    <!--*****Environment values and functions *******-->
    <environment>
<var><type>double</type><name>calcium_level</name></var>
<var><type>double</type><name>substrate_width</name></var>
<functions><file>functions.c</file></functions>
</environment>

    <!--******** X-machine Agent - Keratinocyte ******-->
<xmachine>
<name>keratinocyte</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>int</type><name>type</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>
<var><type>double</type><name>force_x</name></var>
<var><type>double</type><name>force_y</name></var>
<var><type>double</type><name>force_z</name></var>
<var><type>int</type><name>num_xy_bonds</name></var>
<var><type>int</type><name>num_z_bonds</name></var>
<var><type>int</type><name>num_stem_bonds</name></var>
<var><type>int</type><name>cycle</name></var>
<var><type>double</type><name>diff_noise_factor</name></var>
<var><type>int</type><name>dead_ticks</name></var>
<var><type>int</type><name>contact_inhibited_ticks</name></var>
<var><type>double</type><name>motility</name></var>
<var><type>double</type><name>dir</name></var>
<var><type>double</type><name>iradius</name></var>
</memory>

        <!--     Defining functions     -->
<functions>
<function>
<name>output_location_0</name>
</function>
<function>
<name>check_agent_id</name>
<depends><name>output_location_0</name><type>internal</type></depends>
</function>
<function>
<name>cycle</name>
<depends><name>check_agent_id</name><type>internal</type></depends>
```

```
</function>
<function>
<name>differentiate</name>
<depends><name>cycle</name><type>internal</type></depends>
</function>
<function>
<name>migrate</name>
<depends><name>differentiate</name><type>internal</type></depends>
</function>
<function>
<name>output_location_1</name>
<depends><name>migrate</name><type>internal</type></depends>
</function>
<function>
<name>resolve_forces_1</name>
<depends><name>output_location_1</name><type>internal</type></depends>
</function>
</functions>
</xmachine>
    <!--********** End of Agent - Keratinocyte   **************-->

    <!--** Messages being posted by the agents to communicate **-->
    <messages>
        <!--     Message for Cell location      -->
        <message>
            <name>location</name>
            <var><type>int</type><name>id</name></var>
            <var><type>int</type><name>type</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>
            <var><type>double</type><name>dir</name></var>
            <var><type>double</type><name>motility</name></var>
            <var><type>double</type><name>range</name></var>
            <var><type>int</type><name>iteration</name></var>
        </message>
    </messages>
    <!--**** End of Messages ******-->

</xmachine_agent_model>
```

C COPASI data structure definition

```
<?xml version="1.0" ?>
<xmachine_agent_model>
<name>Keratinocyte colony model</name>
<author>Phil McMinn</author>
<!-- Modified by Salem Adra - 11/2007- -->

<environment>

    <var><type>double</type><name>calcium_level</name></var>
    <var><type>double</type><name>substrate_width</name></var>

    <!--**********************COPASI DATA STRUCTURE***********-->
    <datatypes>
    <datatype>
    <name>copasi_data</name>
    <desc>Used by Cells to hold copasi data</desc>
    <var><type>char_array</type><name>metabolite</name></var>
    <var><type>double</type><name>concentration</name></var>
    </datatype>
    </datatypes>
    <!--********************************************************-->

    <functions><file>functions.c</file></functions>

</environment>

<xmachine>
    <name>keratinocyte</name>
    <memory>
    <var><type>int</type><name>id</name></var>
    <!--********************Array of 3 copasi_data**********-->
    <var><type>copasi_data</type><name>copasiData[3]</name></var>
    <!--****************************************************-->
    <var><type>int</type><name>type</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>
    <var><type>double</type><name>force_x</name></var>
    <var><type>double</type><name>force_y</name></var>
    <var><type>double</type><name>force_z</name></var>
    <var><type>int</type><name>num_xy_bonds</name></var>
    <var><type>int</type><name>num_z_bonds</name></var>
    <var><type>int</type><name>num_stem_bonds</name></var>
    <var><type>double</type><name>motility</name></var>
    <var><type>double</type><name>dir</name></var>
    <var><type>double</type><name>iradius</name></var>
    </memory>

    <functions>
        <function>
            <name>output_location_0</name>
        </function>
```

```
        <function>
            <name>check_agent_id</name>
            <depends><name>output_location_0</name><type>internal</type></depends>
        </function>
        <function>
            <name>cycle</name>
            <depends><name>check_agent_id</name><type>internal</type></depends>
        </function>
        <function>
            <name>differentiate</name>
            <depends><name>cycle</name><type>internal</type></depends>
        </function>
        <function>
            <name>migrate</name>
            <depends><name>differentiate</name><type>internal</type></depends>
        </function>
        <function>
            <name>output_location_1</name>
            <depends><name>migrate</name><type>internal</type></depends>
        </function>
        <function>
            <name>resolve_forces</name>
            <depends><name>output_location_1</name><type>internal</type></depends>
        </function>
    </functions>
</xmachine>
<messages>
    <message>
        <name>location</name>
        <var><type>int</type><name>id</name></var>
        <var><type>int</type><name>type</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>
        <var><type>double</type><name>dir</name></var>
        <var><type>double</type><name>motility</name></var>
        <var><type>double</type><name>range</name></var>
        <var><type>int</type><name>iteration</name></var>
    </message>
</messages>
</xmachine_agent_model>
```

D Initialisation of a model using COPASI

```
<states>

<environment>
<calcium_level>1.3</calcium_level>
<substrate_width>510.0</substrate_width>
</environment>

<itno>0</itno>

<xagent>
<name>keratinocyte</name>
<id>1</id>
<copasiData>{{Shc, 0.05}, {Ras, 0.08}, {Erk-PP, 0}}</copasiData>
<type>0</type>
<x>56</x>
<y>321</y>
<z>0</z>
<motility>1.2</motility>
<dir>2.3</dir>
<iradius>3.1</iradius>
</xagent>

<xagent>
<name>keratinocyte</name>
<id>2</id>
<copasiData>{{Shc, 0.03}, {Ras, 0.04}, {Erk-PP, 0}}</copasiData>
<type>0</type>
<x>56</x>
<y>321</y>
<z>0</z>
<motility>1.2</motility>
<dir>2.3</dir>
<iradius>3.1</iradius>
</xagent>

</states>
```

E Results achieved at the 10th iteration for a model using COPASI

```
<states>
<itno>10</itno>
<environment>
<calcium_level>1.300000</calcium_level>
<substrate_width>510.000000</substrate_width>
</environment>

<xagent>
<name>keratinocyte</name>
<id>1</id>
<copasiData>{{Shc, {0.009901}}, {Ras, {0.009901}}, {Erk-PP, {0.000099}}}</copasiData>
<type>0</type>
<x>444.000000</x>
<y>486.000000</y>
<z>0.000000</z>
<motility>1.200000</motility>
<dir>2.300000</dir>
<iradius>3.100000</iradius>
</xagent>

<xagent>
<name>keratinocyte</name>
<id>2</id>
<copasiData>{{Shc, {0.046244}}, {Ras, {0.076244}}, {Erk-PP, {0.003756}}}</copasiData>
<type>0</type>
<x>56.000000</x>
<y>321.000000</y>
<z>0.000000</z>
<motility>1.200000</motility>
<dir>2.300000</dir>
<iradius>3.100000</iradius>
</xagent>

</states>
```

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

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

[7] K.A. Holbrook. Ultrastructure of the epidermis. *Keratinocyte handbook*, pages 3–39, 1994.

[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] S. Hoops, S. Sahle, R. Gauges, C. Lee, J. Pahle, N. Simus, M. Singhal, L. Xu, P. Mendes, and U. Kummer. Copasi a complex pathway simulator. *Bioinformatics*, 22:3067–3074, 2006.

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

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

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

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

[14] 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.

[15] T. Sun, P. McMinn, S. Coackley, M. Holcombe, R. Smallwood, and S. MacNeil. An integrated systems biology approach to understanding the rules of keratinocyte colony formation. *The Royal Society Interface*, pages 1077–1092, 2007.

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

[17] 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.

[18] D. Walker, J. Southgate, G. Hill, M. Holcombe, D.R. Hose, S.M. Wood, S. MacNeil, and R. Smallwood. The epitheliome: agent-based modelling of the social behaviour of cells. *Biosystems*, 76:89–100, 2004.

[19] D. Walker, T. Sun, S. MacNeil, and R. Smallwood. Modelling the effect of exogenous calcium on keratinocyte and hacat cell proliferation and differentiation using an agent-based computational paradigm. *Tissue Eng.*, 12:2301–2309, 2006.

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

Glossary

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.

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.