

Validation and discovery from computational biology models

Mariam Kiran*, Simon Coakley, Neil Walkinshaw, Phil McMinn, Mike Holcombe

University of Sheffield, Department of Computer Science, Regent Court, 211 Portobello Street, Sheffield S1 4DP, United Kingdom

Received 1 January 2008; received in revised form 10 March 2008; accepted 18 March 2008

Abstract

Simulation software is often a fundamental component in systems biology projects and provides a key aspect of the integration of experimental and analytical techniques in the search for greater understanding and prediction of biology at the systems level. It is important that the modelling and analysis software is reliable and that techniques exist for automating the analysis of the vast amounts of data which such simulation environments generate. A rigorous approach to the development of complex modelling software is needed. Such a framework is presented here together with techniques for the automated analysis of such models and a process for the automatic discovery of biological phenomena from large simulation data sets. Illustrations are taken from a major systems biology research project involving the *in vitro* investigation, modelling and simulation of epithelial tissue.

© 2008 Elsevier Ireland Ltd. All rights reserved.

Keywords: Agent-based modelling; Simulation; X-machines; Validation and testing; Parallel computation

1. Introduction

Many natural systems consist of individual entities that interact with each other in relatively simple terms to exhibit a more complex, aggregate behaviour. As an example high-level phenomena, such as wound healing and tumors, are the cumulative result of low-level cellular interactions within the epidermis. Identifying the simple microscopic interactions between the low-level components is crucial for a full understanding of the more complex macroscopic behaviour.

Without the help of computers, modelling approaches have usually involved manually sifting through biological data and deriving differential equations that approximate the 'average' behaviour of the system as a whole. The problem with such approaches is that they fail to capture the large numbers of local interactions between components that are the cause of the high-level behaviour. Consequently research is increasingly turning towards bottom-up modelling approaches, in the hope of building more realistic models that capture these interactions and in the process provide deeper insights into the reasons for the high-level behaviour.

Agent-based modelling has been shown to produce very useful methods and results in the field (cf. Xavier and Foster's work on microbial biofilms [Xavier and Foster, 2007](#)). Models are generated from the bottom-up, constructing for example, molecules or the cells as individual active components, or agents. Once the lowest level agents have been identified, rules that govern their interaction can be generated, which results in molecular interaction models, cell societies, etc. allowing data, structures and functions to evolve using concepts of emergence. The process ties in experimental biology with computational techniques to handle complexity at various levels and makes it easy to process, study and predict conclusions from the available data.

There are various environments which allow modellers to use agent-based modelling to investigate different models. Various frameworks have been released to facilitate this and some of them have been summarized in [Table 1](#). Both [Xavier and Foster \(2007\)](#) and [Railsback et al. \(2006\)](#) provide detailed comparisons of various platforms by implementing similar models on each of them.

Most of these frameworks are designed to be specific to the domains they are applied in, *repat*, for instance is designed for social sciences. Allowing an agent-based framework to be generic enough to allow any kind of agent simulation has been a challenge over the years. In this paper we introduce flexible large-scale agent-based modelling environment (FLAME)¹, a

* Corresponding author. Tel.: +44 114 22 21949.

E-mail addresses: m.kiran@dcs.shef.ac.uk (M. Kiran), s.coakley@dcs.shef.ac.uk (S. Coakley), n.walkinshaw@dcs.shef.ac.uk (N. Walkinshaw), p.mcminn@dcs.shef.ac.uk (P. McMinn), m.holcolombe@dcs.shef.ac.uk (M. Holcombe).

¹ (<http://www.flame.ac.uk>).

Table 1
Comparison of commonly used frameworks

	SWARM	JADE	MASON	RePAST	FLAME
Software methodology	Programmed in objective C, and implemented over a nested structure	Uses FIPA protocols	Programmed in Java and implemented over a layered structure	Programmed in Java	Programmed in C and designed using X-machine approach
Visualisation	3D	3D	3D	2D	2D and 3D
Parallel or serial	Both. Need to wrap objective-C commands in Java for parallel	Both	Both	Both	Both. Uses HPC and MPI message for faster communication
Examples of models executed	Sugarscape, variety from other disciplines	Virus epidemics, sugarscape	Virus epidemics, sugarscape, traffic simulation	Mostly social science projects	Skin grafting, economic models

flexible agent-based framework, which allows agent-based simulations in any domain. Model specifications are based upon formal X-machines (Holcombe and Ipate, 1998). These are generalised state machines that include an internal memory, which provides a mechanism for avoiding many of the state explosion problems that conventional state machines can suffer from. Although traditionally used to specify conventional software systems, our work has applied X-machines to provide a formal basis for the specification of agent-based models of biological systems (Coakley et al., 2006; Sun et al., 2007). These X-machine structures allow agents to carry internal memory data that can be updated as simulations continue.

X-machines enable the specification of agents at a level of abstraction that matches the biologist's conceptualisation of the system. Instead of having to adjust the model to fit the constraints of a particular programming language, systems can be specified in terms of their constituent processes and structures. Once a model is specified, FLAME is able to automatically generate

simulations that incorporate efficient communication between agents and can be executed on a variety of parallel computation platforms. The FLAME framework has been designed from the outset for use on large-scale supercomputers and is capable of efficient simulation of models involving many millions of agents. Simulations executed on FLAME have shown that large numbers of agents can be used with only a minor communication overhead.

Fig. 1 provides an outline of the general modelling process. One factor that is of particular import, and that is central to this paper, is the ability to validate a model and to discover novel aspects of the model's behaviour that might be of interest to the biologist. This is crucial and can be very challenging as the complexity of models increases.

This paper describes a formal, agent-based approach to modelling biological systems. The following section provides a rationale for our work, and particularly our emphasis on validation and the discovery of new facts from our models. Section

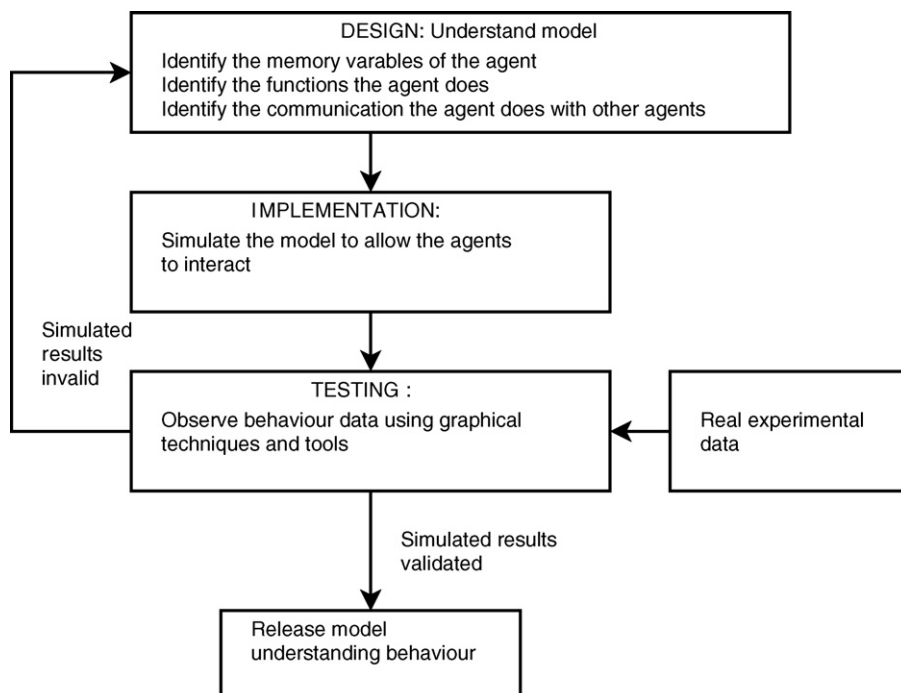


Fig. 1. FLAME modelling process.

3 describes the X-machine modelling approach, and details its implementation with respect to high-performance computers. Section 4 provides a case study, demonstrating the use of FLAME to model the *in vitro* behaviour of skin cells. Section 5 describes our use of an established tool for reverse-engineering properties from software systems to discover novel (and possibly erroneous) properties models and applies it to the skin cell model. Finally Section 6 contains conclusions, along with our future work.

2. Building Trustworthy Scientific Systems

There have been a number of cases where incorrect software has affected the accuracy of the data that has been regarded as correct. Recently (Joosten and Vriend, 2007) large sets of data were withdrawn from the Protein Data Bank repository because the software used to generate the data was seriously flawed.

Building correct software is a challenge on any application domain and there are no easy solutions to the problem. Many important scientific software systems have evolved from experimental pieces of software often developed by scientists with little or no training in software engineering. These systems have become very complex and yet at their core there may be software that is potentially flawed. As programs evolve through multiple versions and technologies the problems get compounded. Software that works correctly on one type of computer platform may behave rather differently on another which has a different architecture. As this happens it may not be obvious that the numbers being generated are wrong without an extensive programme of testing and validation analysis.

The sudden growth of systems biology has raised a number of questions about our ability to understand complex systems through computational modelling—and thus the use and analysis of software. Systems biology combines systems modelling with large-scale experimental biology often using vastly complex bioinformatics data sets. The simulation data produced from models is usually massive and has to be analysed. Information discovery is thus a vital component of systems biology and has received little attention so far. To deal with the complexity of systems biology modern computing techniques are needed: software for model development, making use of advanced software design methods when building models; software for validation, using advanced techniques for validating models; software testing, the model may be right but the implementation may be flawed. The key point is that it's too complicated to do all this by hand.

Hatton (2007) has analysed the source code of a number of scientific applications and found many examples of programmes that were seriously incorrect. In fact, this research poses the question of what can we rely on in terms of using software to understand and predict complex biological phenomena. If some of this data is being used in medical treatments or the analysis of the possible effects of drugs and therapeutic interventions then lives could be put at risk.

Models of biological systems are becoming increasingly complex and this complexity is increasingly difficult to manage. For example, many CellDesigner ([\[biology.org/cd/\]\(http://biology.org/cd/\)\) models are very intricate and contain an overwhelming amount of information. This complexity can often obscure those relationships and features within the model that are particularly important to the biologist. Biological models are usually adjusted over time. They are prone to faults and are often maintained by different people. Isolated changes to a model will often occur in succession, not taking each other into account, and can, as a result, have a degenerative effect on the model as a whole by introducing unforeseen alterations to its behaviour. The overall complexity of the model increases and maybe no-one really understands the model any more.](http://www.systems-</p></div><div data-bbox=)

A more systematic process for model sharing and version control is needed. Once a model exceeds a certain level of complexity, it becomes very difficult to ensure that it is correct. Large systems are arduous and difficult to validate (a process that has to be carried out manually). Although a system can be simulated to ensure that its output conforms to laboratory observations, this is often carried out on an ad hoc basis, and the behaviour of the model is rarely exercised systematically.

Models also need to be described in a more suitable language than the details of source code, something that biologists can understand and that revolves around concepts, metaphors and processes that they understand rather than the intricacies of software development methods.

The basic framework that we discuss in this paper is built around a simple concept that can be used as a basis for describing and defining the essential components in a complex biological model. Each system is conceived to be a collection of, often disparate, entities which we call agents. They may, themselves, be complex systems and so many models will have a hierarchical structure. Each of these agents has its own life cycle from creation to removal and will behave in different ways depending on where it is and what is going on its location and what stage of its life cycle it is in. These factors can be defined using a simple diagrammatic notation and captured into a textual summary based on XML. There will be many types of communication that will go on between the agents and this is also defined—it will influence what each agent does.

The whole model is then aggregated into a software program automatically and the simulations can then be performed. This is described further in the next section and illustrated in Section 5.

3. Developing Correct Models with the FLAME Framework

The increasingly widespread use of agent-based models within the field of systems biology means that more emphasis needs to be applied to their formal verification and validation. As the systems become more complex, simulating them will become increasingly computationally expensive. The FLAME framework has been developed to address these two concerns. It is based on a formal modelling paradigm that is both accessible to the biologist, but also permits the use of well-established state machine analysis and testing algorithms for the sake of verification and validation. It also incorporates a number of mechanisms to ensure that the simulation of the model can be distributed and

executed in parallel on a variety of high-performance computing platforms.

3.1. X-machines and FLAME

Development methodologies proposed for agent-based models are mostly based around abstract representations like UML. For a concise definition, X-machines offer a more formal computational model that specifies the exact data transformations and control flow. State machines have been used extensively in the past to describe the control logic for low-level hardware and software applications.

X-machines (Holcombe and Ipate, 1998) are a generalisation of conventional state machines that permits the complex behaviour of agents to be expressed abstractly, using the familiar notion of states and transitions. However, unlike conventional state machines, X-machines contain a set of processing functions that are tied to each transition and operate upon a global memory. Thus, instead of merely choosing the next state by reading an input symbol, an X-machine chooses its next state by looking at the current state of its memory as well as the current input. The benefits of using this representation are twofold:

1. Complex units of behaviour can be wrapped into “processing functions”, and thus provide a flexible means of abstraction.
2. The basic structure is still in the form of a state machine, which means that it remains intuitive to read, and applicable to established testing and verification techniques.

Formally, X-machines are defined as an 8-tuple $XM = (Q, \Sigma, \Gamma, M, \Phi, F, q_0, m_0)$ where:

- Q is the finite set of states.
- Σ is the alphabet of input symbols (messages that can be received from other X-machine agents).
- Γ is the alphabet of output symbols (messages that can be sent to other X-machine agents).
- M is the possibly infinite set of memory configurations.
- Φ is a set of partial functions ϕ that map an input and a memory configuration to an output and a (possibly) different memory configuration, $\phi : \Sigma \times M \rightarrow \Gamma \times M$.
- F is a partial function that determines the next state, $F : Q \times \phi \rightarrow Q$. This can be described as a state transition diagram, where each transition is labelled by function ϕ .
- q_0 is the initial state.
- m_0 is the initial memory configuration.

In practice, the process of defining an agent using this framework is relatively straightforward. For each agent a state transition diagram is created that outlines the high-level behaviour of the agent in terms of its abstract processing functions. Then, each individual processing function can be defined in detail, in terms of its memory requirements (pre- and post-conditions), inputs and outputs.

Our FLAME framework provides an infrastructure that can take specifications in the form of the X-machines described above, and simulate their behaviour on a large scale. Fig. 2

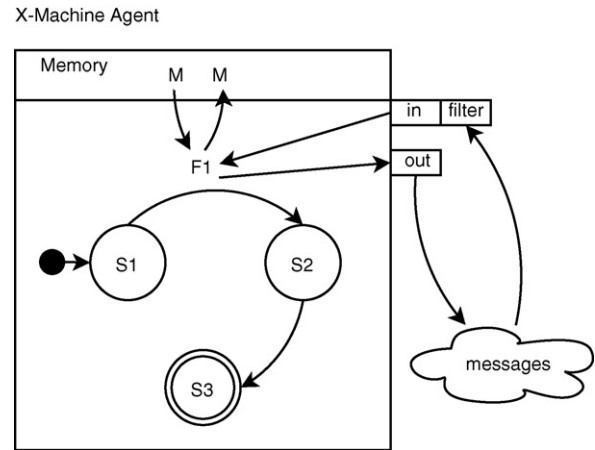


Fig. 2. X-machine agent example.

illustrates how an X-machine agent operates in FLAME. The framework maintains a global list of messages that are sent between agents. An agent in state $S1$ receives some input (belonging to Σ) from another agent and, depending on its current memory state, executes function $F1$ (as defined in F). This results in a new memory state, and an output message (belonging to Γ) is sent to another agent. The entire simulation (which can consist of thousands of agents) is executed in steps where, in a single step, every agent in the system is updated concurrently.

If a particular function in one agent depends on the output of another function in a different agent, a communication synchronisation point is added to make sure that any output messages have arrived for input. In this way a simulation can be spread across separate processors because each executes the same order of agent functions and synchronises communication at the same time.

On a large scale, certain models can depend on a substantial amount of communication between individual agents. If at every iteration all of the agents have to process the entire list of messages, this can result in a communication bottleneck, and impact on the overall performance of the system. The FLAME framework ameliorates this by introducing filters. These can be attached to each agents and ensure that it only processes relevant messages. As an example, a filter in a cellular simulation might be that a particular cell can only be affected by messages that are sent by other cells in a particular radius.

Once the conceptual model of the system has been generated (in terms of X-machines), it has to be translated into a format that can be directly executed. This is done automatically by FLAME. The modeller can provide an abstract X-machine description of their model, and FLAME automatically translates it into C code that can be compiled to be run on the desired computing architecture.

3.2. Scalability on High-Performance Computers

Agent-based models can consist of thousands of agents, and the process of simulating their behaviour on a large scale can be computationally very expensive. Although the expense is in many ways inevitable (there is an essential amount of processing

that needs to be carried out), it can be substantially reduced by the use of filters (as described above), along with the fact that agent-based models can be largely executed in parallel. The FLAME framework was constructed with the aim that models should be executable on a wide range of high-performance computing architectures. This should however not require any specialist intervention from the modeller.

Compatibility with a wide variety of high-performance architectures was ensured by using the Message Passing Interface (MPI) (Foster, 1995) framework for C. When the models are translated into C, they adhere to the MPI interface. The communication synchronisation points ensure that concurrently executing agents remain in sync with each other.

To establish that the framework is scalable and can be used for large scale models a model was built to provide benchmarks for performance on a variety of supercomputers. The model is set on a two-dimensional plane, where each agent represents a randomly sized circle, and the circles overlap. Each agent has to ensure that it is not overlapping with other agents at the end of each iteration, moving into free space where necessary. This was executed using (one million randomly placed circles) on the following high-performance systems (with the help of our colleagues at the Science and Technologies Facilities Council):

- SCARF: A cluster with 360 2.2 GHz AMD Opteron processor cores and 1.296TB total memory. Communication includes gigabit networking and a Myrinet low latency interconnect.
- HAPU: An HP cluster with 128 2.4 GHz Opteron cores, with 2 Gb memory per core, and a Voltaire InfiniBand interconnect.
- NW-GRID: A cluster with 32 SUN x4100 server nodes containing 2 Dual Core 2.4 GHz Opterons with 8 Gb of memory each.
- HPCx: An IBM cluster based on the pSeries 575 system made up of 160 clusters of 16 1.5 GHz POWER5 processors and 32 GB memory. Each cluster is connected via IBM's High-Performance Switch (HPS).
- MANO: An IBM Blue Gene/L machine comprised of 1024 nodes of dual-core 700 MHz PowerPC chips with the second CPU usually dedicated IO and communications.

The results are shown in Fig. 3. The number of processors is shown on the *x*-axis, and the amount of time consumed is shown on the *y*-axis. The results show that substantial gains are made from 1 to 49 processors, and that the increase reduces somewhat with higher numbers. This is perhaps more of a reflection on the characteristics of the circle problem than FLAME; different benchmarks resulted in similar scales of reduction, but varied in terms of the number of processors that were required. Ultimately, FLAME successfully compiles the X-machine models in such a way that they can be efficiently distributed to run in parallel.

3.3. Providing Input X-machine Models and Visualising Output

The main rationale for using X-machines to model biological systems is that they provide an intuitive means to specify rela-

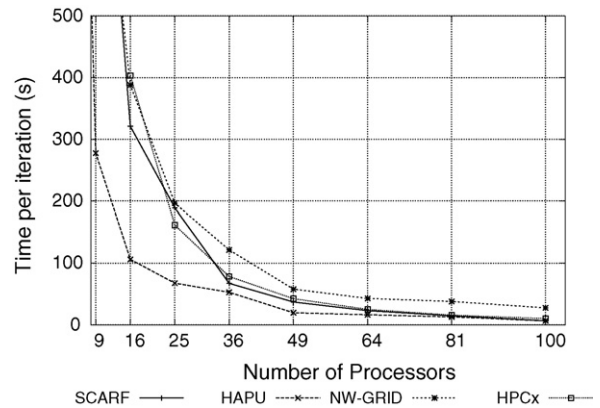


Fig. 3. Benchmarks from high-performance computers.

tively complex systems. This is particularly important because the biologists should be able to reason in terms of the model, and not need to be concerned with the low-level details. In this respect, it is particularly important that there is a straightforward means to present FLAME with the model, and that the output of the model is presented in a way that is intuitive to the modeller.

FLAME currently reads a model in as an XML file (this is described in more detail by Coakley et al. (2006)). XML is, like MPI, widely supported, which makes it easy to specify user-friendly X-machine editors. This facilitates the generation of specifications in specific domains, and is part of our ongoing work.

Understanding and interpreting the output of agent-based simulations is a particularly challenging area. Every iteration, the agents change their state, which results in a potentially vast number of state trajectories. For models that deal with location and cellular kinetics, FLAME includes a three-dimensional visualise that permits the biologist to interact with it (zoom in and out, or rotate). Some sample screen shots appear in Section 5, which also elaborates on some alternative techniques that can be used to explore and interpret the output data.

4. Case Study: Keratinocyte Model

As part of the Epitheliome project at Sheffield, Sun et al. (2007) have developed a model of the *in vitro* behaviour of skin cells using FLAME. The ultimate application is the development of methods to produce reconstructed human skin for patients with heavy skin loss—for example through chronic burns, wounds or skin disease.

The aim of the model is to understand how cells proliferate and organise themselves into layers of skin tissue through colony formation. The model is biologically useful, because it can predict, on the basis of the colony epicentres, how or whether a wound will heal. This is because a colony's maximal size is bounded. Therefore, if the remaining colony epicentres are too far apart after the wound, the gap will never close. The colony model also explains the morphology of the healing process—i.e. why the two sides of a scratch wound do not meet in straight edges, but rather with 'bumps' caused by circular outgrowths from colony epicentres.

As the main purpose of the model was to learn about cell colony formation, many complex elements of the natural biology have been abstracted away. A simple physical model is used, along with a simple approach to cell cycling and there are no complex cell signalling mechanisms. Each cell is simply modelled as sphere on a virtual culture dish. In our model, cells stay at a constant size and shape.

Each cell agent stores its location co-ordinates in its local memory, and behaves according to programmed rules for movement and cell division, as described below. There are four types of cell: stem cells, transit-amplifying (TA) cells, committed cells and corneocytes. Stem cells are found in the centre of the colony, are fairly static and proliferate so long as there is space to do so. TA cells can also proliferate as long as there is space, but can also migrate around the culture plate, depending on the ambient levels of calcium. Committed cells are TA or stem cells that have undergone a differentiation process, they can no longer proliferate, corneocytes are dead cells that are found on the top levels of skin tissue.

4.1. Physical Model

The physical model simply applies a force to overlapping cells to separate them. The physical model is also built so that different cells exert different forces, for example stem cells bond strongly to themselves and the culture plate (which is why they remain static), as do committed cells and corneocytes.

4.2. Cell Cycle

Each stem and TA cell follows a cell cycle whereby they divide after a pre-determined period, i.e. a stem cell will produce two daughter stem cells, and a TA cell two daughter TA cells. If a cell is 'contact-inhibited', that is, there is no space around to divide it goes into a special dormant state of the cell cycle referred to as G0.

4.3. Differentiation

Differentiation is the process whereby a cell changes from one type to another. Initially stem cells divide and cluster (Fig. 4). When the cluster reaches a certain size however, cells on the cluster edge differentiate into TA cells. In practice this rule is coded by each stem cell scanning its vicinity. If it cannot find a certain number of fellow 'mate' stem cells within the range d_1 , it differentiates. When TA cells are a distance d_2 away from the stem cell epicentre of the colony, they differentiate into committed cells (Fig. 5). Stem and TA cells that have also been in G0 for a certain period of time also differentiate into committed cells (Fig. 6). Finally a committed cell will 'die' and become a corneocyte after a further period of time (Fig. 7).

4.4. Stratification

Skin cells self-organise into layers. In the model, daughter cells are pushed up a layer if there is no lateral space (Fig. 8).

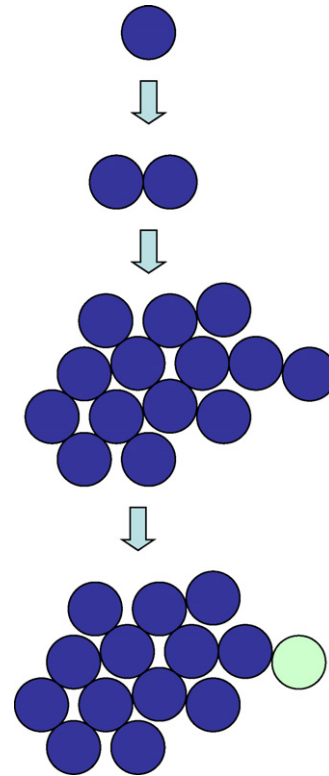


Fig. 4. Differentiation from stem cells to TA cells. Stem cells are blue and TA cells are green.

The model was investigated in terms of a number of biological issues. In particular, following extensive simulations, the prediction was made that for normal human keratinocytes the position of stem cells would influence the pattern of cell migration post-wounding. This was then confirmed experimentally using a scratch wound model (Sun et al., 2007). However, there may be many other interesting properties that the model possesses that would be of biological interest, some, perhaps, unexpected. A more systematic method is needed to analyse the model behaviour.

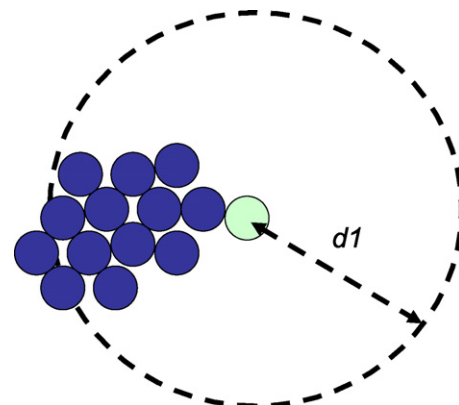


Fig. 5. Mechanism for stem to TA cell differentiation.

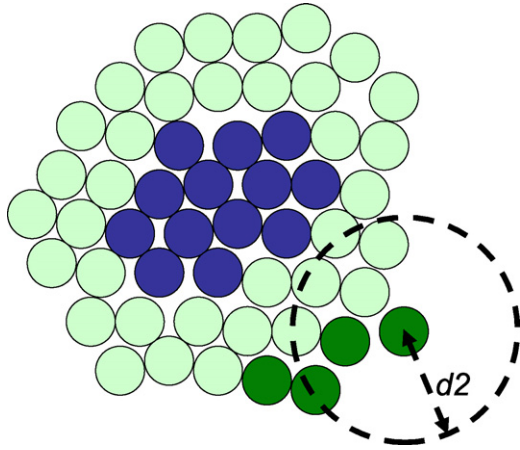


Fig. 6. TA cells now dominate the edges of the colony. As the colony gets bigger TA cells far away from the stem cell epicentre differentiate into committed (dark green) cells.

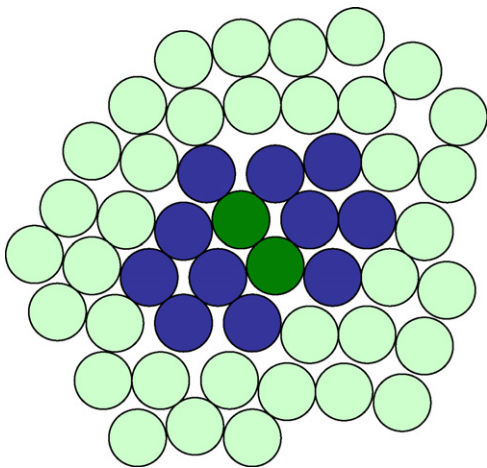


Fig. 7. Stem and TA cells that have been contact-inhibited for a certain period of time differentiate into committed cells.

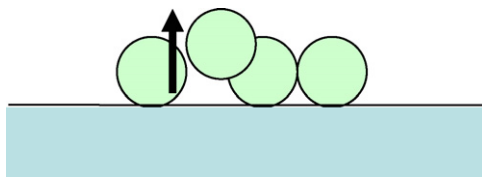


Fig. 8. Stratification.

5. Discovering Properties of Biological Models

Although it is straightforward to specify at a microscopic level (i.e. individual agents), the vast numbers of agents and potential agent interactions makes emergent behaviour difficult to understand, validate and maintain.

Although emergent behaviour is easily observable at a macroscopic level (e.g. we can see cells clustering into colonies via a simple visualisation), it is difficult to understand, in terms of individual agents, why the clustering is occurring. What are the (interesting) aspects of system behaviour that are pertinent to this behaviour? Manually identifying these from the usually

vast domain of potential agents and agent interactions can be extremely challenging.

This can in turn lead to problems when validating the model. Although the behaviour of the system might seem valid superficially (i.e. a visualisation might seem to behave as expected), it is again difficult to ensure that this is the case. The major benefit of agent-based modelling (that complex collaborative behaviour can be elicited from simple individuals) is a double-edged sword; small faults in agents can equally amount to gross misbehaviour. As a simple example, during the development of the skin cell colonies described in Section 4, one version of the model (erroneously) started producing large tumorous clusters of cells, simply because of the fact that the modeller had forgotten to specify that virtual cells should adhere to the virtual Petri-dish.

Even if the model behaviour is correct to begin with, ensuring that it remains so becomes increasingly challenging as it evolves. Models (especially larger ones) tend to be maintained and developed by teams of modellers and developers, who can often introduce conflicting changes to the model. This is exemplified by the BioModels database (<http://www.ebi.ac.uk/biomodels/>), where many of the models have a long history of modifications, submitted by peers in the community. Isolated changes to the model will often occur in succession, not taking each other into account, and can as a result have a degenerative effect on the model as a whole. This compounds the aforementioned problems of comprehension and validation.

These problems are not specific to agent-oriented models, or even the discipline of computational biology as a whole. They are analogous to the problems that arise during software development, where they have been studied for decades. As is the case with biological models, software implementations rapidly become too intricate to understand in their entirety. Software, too, is often validated on a superficial basis. Like BioModels, a number of large software repositories exist that enable multiple developers to concurrently add lots of changes to software systems, thus introducing the same problems of degeneration through evolution that occur with biological models.

The most important component of any (software engineering) solution to the aforementioned problems is the use of abstract specifications. These can be used for documentation, to make it easier for developers to communicate and understand the system. They can be used as the basis for a number of powerful validation and verification techniques to ensure that changes to the system remain consistent with the requirements. Most importantly, a number of techniques have been developed (e.g. Ernst et al., 2001) that attempt to extract these specifications directly from an implementation, so that only a minimal amount of effort needs to be invested by the developer, but they can still take advantage of the benefits.

5.1. Specifying Systems with Invariants

Invariants are means of formally specifying the behaviour of a software system. They provide a mathematical description of what is expected of the system at particular points during its execution, without going into the procedural details of how this is achieved. To specify a system with invariants, it is divided into

its constituent functions, and each function is annotated with its respective invariants. The following types of invariant can be used to annotate a function:

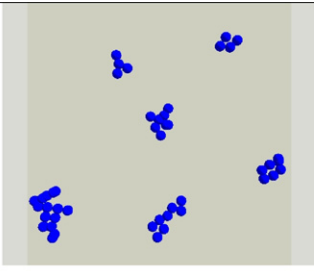
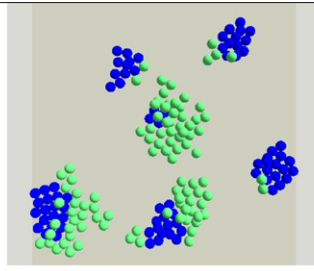
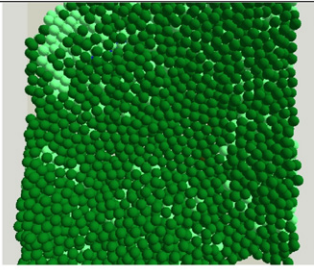
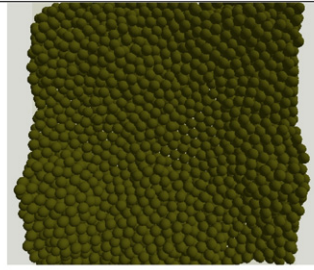
- **Pre-condition:** Specifies the conditions that must hold between certain variables before the function is executed.
- **Post-condition:** Specifies the conditions that must hold between certain variables once the function has finished executing. Together, the pre- and post-conditions summarise the effect that a function has on the state of the program as a whole.
- **Scoped invariant:** Specifies the relationships that must hold between certain program variables within the scope of a program element such as a function, loop, or object (in an object-oriented program).

A program that is annotated with such invariants becomes amenable to a host of powerful analysis techniques that can be used to address the problems mentioned above. Invariants are

useful for program comprehension and documentation, because they specify the essential, required behaviour of the system without delving into the details of its implementation. Because it is easier to understand a program that is annotated with invariants, it becomes easier to validate by inspection. As the program evolves as a result of bug-fixes and other changes, it is straightforward to analyse, using a host of simple checkers that ensure that the essential functionality of the software is not affected, and that the fix to one fault does not in turn introduce a new one.

In practice, software engineers rarely annotate their code with invariants. As the system evolves, the invariants are often not kept up to date. Ernst et al. (2001) accept that developers do not routinely produce specifications, and devised a technique to reverse-engineer the specifications from the software system itself. Their Daikon technique observes the software as it is executed (multiple times), and records the values of variables at particular points in the program (e.g. the entry and exit points of methods to record pre- and post-conditions).

Table 2
Selected properties from Keratocyte model

	
<p>Stem cells (blue):</p> $0 \geq x \geq 500$ $27.3 \geq y \geq 467.95$ $z = 0.0$ $0 \geq distanceTravelled \geq 840.529$ <p>$(distanceTravelled = 0) \rightarrow (force_x = 0)$ $(distanceTravelled = 0) \rightarrow (force_y = 0)$ $(distanceTravelled = 0) \rightarrow (dir = 0)$</p>	<p>Transit Amplifying (TA) Cells (light green):</p> $0 \geq x \geq 500$ $0 \geq y \geq 500$ $0 \geq z \geq 43.68$ $3.015 \geq distanceTravelled \geq 1959.867$ <p>$(force_x = 0) \rightarrow (z = 0)$</p>
	
<p>Committed Cells (dark green):</p> $0 \geq x \geq 500$ $0 \geq y \geq 500$ $0 \geq z \geq 79.45$ $0.02 \geq distanceTravelled \geq 758.99$	<p>Corneocyte Cells (brown):</p> $0 \geq x \geq 500$ $0 \geq y \geq 500$ $0 \geq z \geq 84.56$ $0.02 \geq distanceTravelled \geq 143.44$ <p>$(dir = 0) \rightarrow (z = 0) \rightarrow (motility = 0)$</p>

Daikon operates by attempting to fit relationships between variable values at each point to a set of predefined rules (invariants). These rules hold true for every execution that was observed. If they are correct, they can be inserted into the program in the form of assert statements to ensure that they are not broken as the software evolves. Besides the benefit with respect to preventing the introduction of new errors during evolution, it has also become a powerful basis for automated unit testing.

5.2. Discovering Invariants of Biological Systems

The ability to automatically reverse engineer invariants is just as valuable with biological models as it is with software systems. Such invariants can help the modeller to understand and discover potentially erroneous or even novel aspects of system behaviour. Unlike software systems, biological models tend to be experimental; they are not designed to fit a well-designed purpose, but rather to investigate high-level behaviour that may not yet be known to the modeller. What is therefore particularly valuable from this perspective is the potential for Daikon to make explicit this novel behaviour, by exposing any latent relationships between (seemingly) unrelated variables across the system.

Invariants of biological models are discovered in a similar manner to invariants of software systems; the model is simulated multiple times, and variable values are recorded at particular points. Given the set of traces, Daikon identifies those rules that are satisfied by all of the traces at the recording points. It is up to the modeller to select the set of variables that should be analysed, along with the set of points during the execution at which they should be recorded. If the task is to find novel, hidden relationships between variables, it makes sense to include a large range of variables in the analysis, and to record them at frequent intervals. If on the other hand the task is to debug or investigate the model with respect to a specific aspect of behaviour, it makes sense to only select a few relevant variables, and to sample them at the relevant points. This can result in a more focussed set of invariants that are more likely to be relevant.

We have used Daikon to investigate the behaviour of McMinn's Keratinocyte model (Sun et al., 2007), introduced in Section 4. The output from the simulation was processed by Daikon, which suggested 297 invariants that govern the relationships between the various variables in the model. The invariants were particularly good at summarising the movement of the different cell types; a selection is discussed below to provide an idea of the sort of rules that are reverse engineered.

Table 2 shows a small selection of the identified invariants for each of the four cell types, along with a snap shot from the phases in the simulation during which the particular cell types were at their most active. In all cases the x and y coordinates of the cell fitted within a 500×500 area (which was the size of their virtual "Petri-dish"). One interesting rule is that, for stem cells, $z = 0$ is an invariant, which means that stem cells never leave the dish—they are always on the base layer (this rule was not coded in to the model, but reflects correct behaviour). Stem cells start from a random, stationary position

in the dish, and Daikon correctly inferred that, for the cell not to have travelled there cannot have been any force exerted on it, and its direction cannot have changed. The invariants show that TA cells are much more motile than any of the other cell types, accounting for the eventual even distribution across the dish. Finally, perhaps one of the most interesting observations is the semi-layered epithelial structure. Stem cells are restricted to the base, along with TA cells these divide and push TA cells up to $43.68 \mu\text{m}$. Committed cells (post-mitotic cells) get pushed up further (up to $79 \mu\text{m}$) by the dividing stem and TA cells. Finally, the top layer of the skin consists solely of corneocyte cells, which are constantly pushed to the surface by the committed cells below.

6. Future Work and Conclusions

The FLAME framework has been used to model a wide variety of biological systems, particularly epithelial tissue (Coakley et al., 2006; Walker et al., 2006; Sun et al., 2007). These have, coupled with laboratory experiments, led to the discovery of a number of novel facts about the behaviour of skin tissue cells. Current work aims to extend and integrate these models with each other.

The framework provides a mechanism for the discovery of properties of models through the analysis of the simulations which could be either faults in the model or new biological insights. The use of this approach has been of great value in the analysis of several complex biological systems, including the models of epithelial tissue. In our future work we aim to investigate the use of other potential discovery techniques to provide further insights into model behaviour.

Acknowledgements

We thank our colleagues (Chris Greenough, David Worth, Shawn Chin) at the Science and Technology Facilities Council UK for helping to establish the performance of our models on high-performance computers. Mariam Kiran and Simon Coakley are funded through the EU project EURACE, Neil Walkinshaw through the EPSRC project AutoAbstract and Phil McMinn was funded for part of the work through the EPSRC Epitheliome project. Experimental work carried out by Sun Tao was funded under the Epitheliome Project.

References

- Coakley, S., Smallwood, R., Holcombe, M., From Molecules to Insect Communities—How Formal Agent Based Modelling is Uncovering New Biological Facts, *Scientiae Mathematicae Japonicae Online*, e-2006, 765–778.
- Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D., 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.* 27 (2), 99–123.
- Foster, I., 1995. *Designing and Building Parallel Programs* (Online). Addison-Wesley, ISBN 0201575949 (Chapter 8, Message Passing Interface).
- Hatton, L., To what extent can we rely on the results of scientific computations? Keynote talk, Proceedings of the ACAT'07, available at: http://www.leshatton.org/Documents/ACAT07_April2007.pdf.

- Holcombe, M., Ipaté, F., 1998. *Correct Systems: Building a Business Process Solution*. Springer-Verlag.
- Joosten, R., Vriend, G., 2007. PDB Improvement Starts with Data Deposition Science, 317(13 July(5835)), 195.
- Railsback, S.F., Lytinen, S.L., Jackson, S.K. Agent-based simulation platforms: review and development recommendations, *Simulation*, 82(9), 609–623. Online: <http://sim.sagepub.com/cgi/content/abstract/82/9/609>.
- Sun, T., McMinn, P., Coakley, S., Holcombe, M., Smallwood, R., MacNeil, S., 2007. An integrated systems biology approach to understanding the rules of Keratinocyte colony formation. *J. R. Soc. Interface* 4 (17), 1077–1092.
- Walker, D., Wood, S., Southgate, J., Holcombe, M., 2006. An integrated agent-mathematical model of the effect of intercellular signalling via the epidermal growth factor receptor on cell proliferation. *Journal of Theoretical Biology* 242 (October 7(3)), 774–789.
- Xavier, J.B., Foster, K.R., 2007. Cooperation and conflict in microbial biofilms. *PNAS* 104 (January 16(3)), 876–881.