Review

Simulating complex intracellular processes using object-oriented computational modelling

Colin G. Johnson [a,*] Jacki P. Goldman [a,b] William J. Gullick [b]

[a] *Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, England*

[b] *Department of Biosciences, University of Kent, Canterbury, Kent, CT2 7NJ, England*

Keywords: computational modelling; object-oriented methods; intracellular processes; protein-protein interactions.

Abstract

The aim of this paper is to give an overview of computer modelling and simulation in cellular biology, in particular as applied to complex biochemical processes within the cell. This is illustrated by the use of the techniques of *object-oriented* modelling, where the computer is used to construct abstractions of objects in the domain being modelled, and these objects then interact within the computer to simulate the system and allow emergent properties to be observed. The paper also discusses the role of computer simulation in understanding complexity in biological systems, and the kinds of information which can be obtained about biology via simulation.

Contents

1	Introduction	3
2	Varieties of complexity	3
2.1	Complexity as density of information	4

* Author for correspondence
Email address: C.G.Johnson@kent.ac.uk (Colin G. Johnson).

2.2 Structure of information and levels of description 4

2.3 Phase transitions, the "edge of chaos" and small worlds 5

2.4 Non-linearity, hysteresis, catastrophe 7

3 Motivation for using computer modelling 10

3.1 Modelling as a pre-experimental step 11

3.2 Looking inside models 11

3.3 Combining models and emergent phenomena 12

3.4 Modelling and sufficiency 13

3.5 Bringing theory and experiment closer together 13

4 Examples of computer modelling of intracellular processes 14

5 The modelling and simulation process 15

6 An introduction to object-oriented methods 18

6.1 Implementation complexity and problem decomposition 18

6.2 Classes and objects 19

6.3 The OO analysis and design process 21

6.4 Relationships between classes 22

6.5 Implementing object orientation. 24

6.6 Design and implementation techniques: patterns, structures, algorithms 26

6.7 Summary 27

6.8 Object-oriented methods for science 27

7 Simulating intracellular processes 27

8 Conclusions and prospects 33

8.1 On object-oriented methods in general 33

8.2 On OO programming languages 33

8.3 On notations to support OO design 33

8.4 On OO methods in science 34

9 Acknowledgements 34

1 Introduction

Biochemical activity within cells exhibits many kinds of complexity. One way of understanding these forms of complexity is by the use of computer modelling and simulation. The aim of this paper is to outline these different kinds of complexity and give an overview of computer modelling and simulation methods which help us to understand these complexities. The focus is on the techniques of *object-oriented modelling*. These techniques are particularly well suited to scientific modelling, as they allow the creation of computational models of the individual components in a system, and facilitate the interaction between those models to produce an overall simulation of the system.

The paper begins with a discussion of the notion of *complexity*, considering the varieties of complexity, how these various complexities are exhibited in intracellular systems, and how computational methods can help to understand these complexities. The following section discusses ways in which computational modelling fits into the scientific process, and this is illustrated in the next section, which discusses a number of systems which model reality at various levels of detail. The next two sections focus on the details of the computational modelling process, firstly looking at the modelling process, and secondly the tools and techniques required for such modelling using the object-oriented methodology. This is supported by a number of examples. The paper concludes with some pointers to future work in this area.

2 Varieties of complexity

It is commonly suggested the the processes within cells are *complex*. In particular *signal transduction*, *metabolic pathways* and *gene expression* provide examples of complex biochemical processes. However the notion of complexity contains a number of facets. This section considers these notions of complexity, with particular reference to intracellular processes, and how computer modelling can assist in understanding the various kinds of complexity.

2.1 Complexity as density of information

One notion of complexity is that of *density of information*. For example if we consider the epidermal growth-factor system and related systems in several organisms we observe such complexity. In *C. elegans* there is a single ligand type and a single receptor type; in *D. melanogaster* there are five types of ligands but only one type of receptor; however in *H. sapiens* there are 10 ligand types and (effectively) 7 types of receptors (including splice variants).

These increases in the amount of information are typically well handled in computer simulation. This contrasts to experimental techniques, where different molecule-types can require different experimental techniques. In particular the techniques of *inheritance* and *prototyping* in object-oriented modelling (discussed below) provide mechanisms for specifying the general behaviour of a large number of different types of entities, whilst the specific details of a particular entity-type can be specified just for that one.

2.2 Structure of information and levels of description

Another kind of complexity arises from the structure of information within a system. To continue with the discussion from the previous section, consider the relationship between the growth factors and pairs of receptors in the EGF system (Yarden and Sliwkowski, 2001; Gullick, 2001). In this system certain receptor will bind to certain ligands, and some of these will form stable pairs.

One way in which this system is complex is that the matching between ligands and receptor-pairs is (effectively) *arbitrary*. Clearly at one level of description (the binding between the various 3-dimensional structures involved) this mapping is not arbitrary, and can be explained and (in principle) predicted. However at the level at which the system *behaves*, the mapping acts in an arbitrary fashion. This understanding has come to be known only by exhaustive experimental work. It is possible that a behaviour-level explanation of why some bindings occur and some do not may be discovered; a similar change of perspective with regard to the DNA $\rightarrow$ amino-acid triplet coding has occurred. An early understanding of this mapping suggested that it was a "frozen accident", i.e. that the first working code had been fixed because variations in the code would be unable to get an evolutionary foothold. However more recent work (Liebovitch et al., 1996; Hornos and Hornos, 1993; Freeland and Hurst, 1998a,b; Freeland et al., 2000) suggests that evolutionary pressures have shaped the code used, e.g. that the code is optimized so that genotypic errors have minimal effect on phenotype. Nonetheless the possibility for regularities on one level of description to lead to arbitrary irregularities at another

remains possible, and is a potential source of biochemical complexity.

This difference in the amount of structure when a system is viewed at different levels is characteristic of this type of complexity; whilst there clearly are consistencies at one level (in this case the atomic interactions level), these do not follow through to give consistencies at another level (in this case the level of protein-protein interactions).

A related kind of complexity is given by the constraints that molecular structure places on the kinds of phenomena which can be produced. In an artificial system it is common for the possible properties of the system to be distributed uniformly through the space of possible structures; it is usually possible to make a small, continuous change to achieve a desired result. However molecular structure places constraints on what can be achieved in biochemical systems; for example a small number of phosphorylation sites may be accessible, putting constraints on the kinds of interaction which can be achieved without a larger-scale change in the system.

2.3 Phase transitions, the "edge of chaos" and small worlds

A variety of complexity of particular relevance to cellular biology due to its influence on network structure is that exhibited by phase transition and "edge of chaos" phenomena. In many systems the *effective complexity* (i.e. the complexity at the level at which the system is acting) reaches a peak and then tails away with increasing complexity of the underlying system. This is illustrated in cartoon form in figure 1. The simple drawings to the left of the picture have low structural complexity in terms of the information required to describe them, and the impact they make on our eyes has low complexity. The central figure is complex both from the point of view of the structure and in terms of impact. However the right-hand figures are effectively identical in terms of impact, despite the details being very different; similar pictures could be substituted without changing the impact of the image. Things such as the central pictures are often described as being on the "edge of chaos". If you make them more simple, the description becomes very similar at all levels, but if you make them more complex, then it becomes difficult to distinguish between different examples (the *effect* is the same despite the details being different).

An example of this is *phase transition* behaviour. Physical phase transitions are a well-known phenomenon. However the underlying dynamic process can be found in a number of situations. In general a phase transition occurs when small changes in a parameter typically make little change to the observed macroscopic behaviour of the system, but where for a certain small parameter

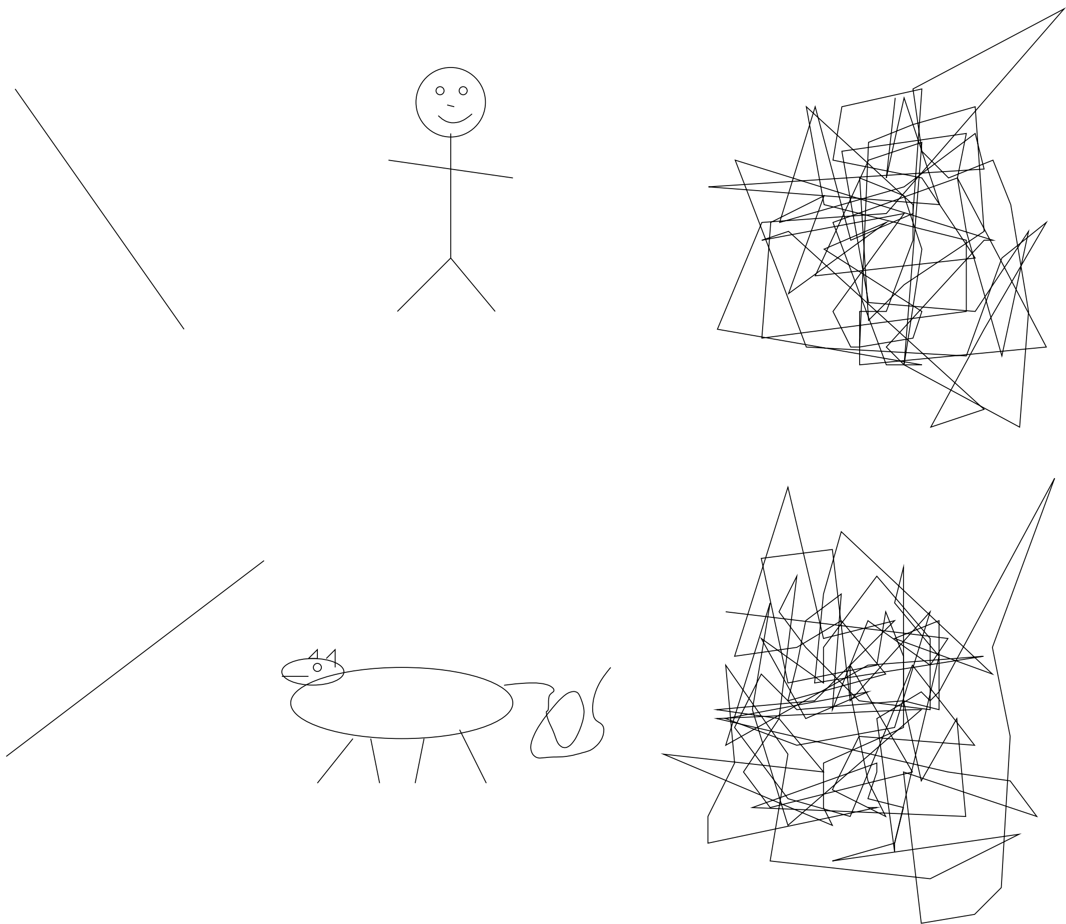


Fig. 1. “Edge of chaos” phenomena illustrated in cartoon form.

region there is a large behaviour change for small parameter change. Biochemical examples of this are the accumulation of influences which give rise to a sudden apoptotic event and the axon firing process in neurons. A somewhat abstract example comes from considering random networks. Consider a set of points, and a probability of linking those points. For low values of the probability the overall connectivity of the network (measured by the largest connected component) remains small as the probability is increased. However at a certain point the network changes from being a collection of sparsely connected clusters into a structure where most of the points are connected into a single component (figure 2). This change occurs for a very small increase in the probability.

A network phenomenon of similar flavour is the "small worlds" phenomenon (Watts, 1999). This is informally known as the "six degrees of separation" phenomenon, as all people on Earth are alleged to be connected by at most six intermediate acquaintances. Watts has pointed out that many complex communication systems exhibit a structure where the number of links is small (i.e. when compared to all other possible ways of organizing the network) compared to the average path-length between randomly chosen points. A canonical example is the organization of human society; it has been suggested that any pair of people in the world are linked by a small number of intermediate acquaintances, despite each person only being acquainted with a small number of others (relative to the whole population). An example of a structure which exhibits this tradeoff is the "connected caveman" structure (figure 3) where small groups of tightly connected nodes are connected via a small number of connections to other such groups. This provides a short path between any pair of nodes in the network at the cost of only a small number of links. It is perhaps unsurprising that evolution has found a use for efficient systems such as these; it has been demonstrated by Wagner and Fell (2001) that certain kinds of metabolic pathways demonstrate this small-world behaviour.

2.4 Non-linearity, hysteresis, catastrophe

An additional source of complexity in biochemical systems is offered by various kinds of *non-linear* behaviour. Broadly non-linearity is where the result of some process varies in a way which is not directly proportional to the action which causes it. There are many different types of such non-linearity: systems where the amount of output increases as a smooth but non-proportional amount of the original signal, critical points where the behaviour changes radically for a small change in some parameter which otherwise causes a near-proportional change, hysteresis effects where a different response is caused by an increase in some parameter and a decrease in the same parameter across the same range.

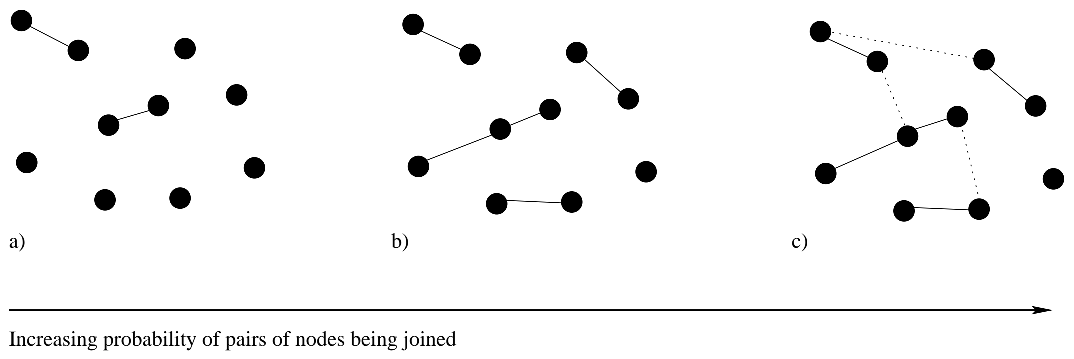


Fig. 2. Phase transitions in random networks.

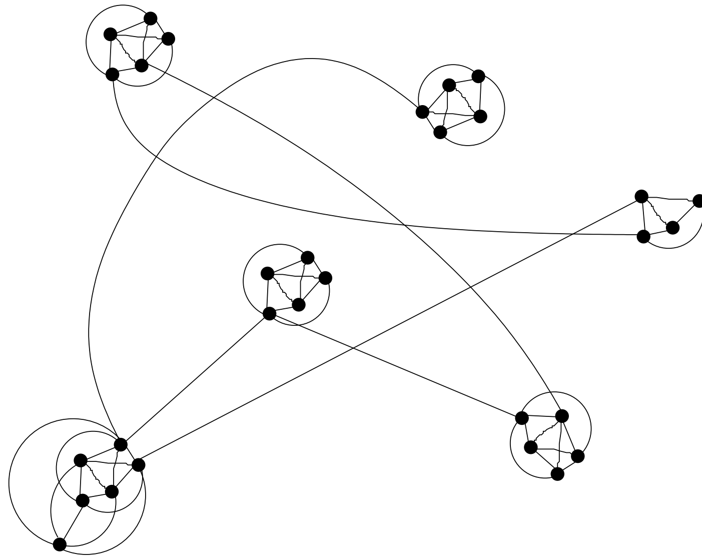


Fig. 3. An example of a network with small-world behaviour—the connected caveman network (after Watts (1999))
.

A simple example of how micro-level behaviour can give rise to simple nonlinearities is given by Alberts et al. (2002, p. 849). This contrasts the behaviour of two receptor types involved in gene regulation. The receptors for the first target gene, conalbumin, increases the activity by an equal amount for each receptor which has bound ligand. The second target gene, ovalbumin, requires a pair of receptors to bind before there is an increase in activity. This leads to a non-linear increase in the response per number of activated receptors.

The following (theoretical) system illustrates hysteresis. As with the ovalbumin, a pair of ligand-bound receptors are required to activate signal at a certain level; however if the receptor remains active until *both* ligand molecules have become unbound, then there is a different behaviour when the system is losing ligand contrasted to when it is gaining.

An interesting consequence of this is given by Ricard (1999), who notes that in such systems it is possible to ascertain whether the concentration of such a protein is increasing or decreasing by taking a sample at a *single* time point. Its position on the hysteresis curve at that point can indicate whether the concentration is increasing or decreasing.

Another example is given by Bray et al. (1998); Bray (1998, 2002). This is concerned with systems of biochemical receptors which must adjust their *sensitivity*, i.e. the system needs to respond to changes in concentration, however it needs to respond to these changes at various levels of ambient receptor concentration. One model explored to explain this is a notion of "cross-talk" between receptors; in addition to reacting to environmental stimulus, the receptors are able to signal each other, and the amount of this inter-receptor stimulation adjusts on a medium-term timescale to the ambient level of receptor in the environment.

3 Motivation for using computer modelling

There are a number of applications of computers in cellular biology. This paper is concerned with the use of computers (and computational techniques) as tools for *modelling* and *simulations* of complex behaviour within the cell. Examples of the kind of phenomena of interest are *signal transduction*, *metabolic pathways* and *gene expression*. Other applications to computers in cellular biology (e.g. the processing of microscope images, the statistical processing of experimental data) and computations which are concerned primarily concerned with the details of interactions between pairs of proteins in the cell at the micro-level (see e.g. Alonso et al. (2002); Smith and Sternberg (2002)) are not considered. Also the vast amount of work in this area within the specialized domain of neuroscience is also outside the scope of this review.

There are a number of motivations for the application of computing in cellular biology, and a number of different types of results which can be extracted from such application. This section explores these various roles for computer simulation.

3.1 Modelling as a pre-experimental step

One reason for using computer models is as a method of *experimental design* and *choice of experiments*. Laboratory work is time-consuming and often expensive, and a simplified computer model can be used prior to laboratory work for a number of reasons. Firstly it allows a basic testing of the feasibility of a particular hypothesis to see whether it produces results which are roughly as expected. A second, related, motivation is in deciding what needs to be measured in an experiment. A computer simulation can be created based on hypotheses about how the system works. This simulation can be repeated many times, slowed down, et cetera, so that the right type of observation can be made when the real experiment is done.

One currently underexplored application is the use of computers to search for interesting, unknown phenomena worthy of investigation. Recent work in the data-mining area (Freitas, 1999) has been concerned with computer-based studies of data for patterns which are likely to be surprising to observers. These ideas have been applied to science, in particular in pure mathematics (Colton et al., 2000a,b).

3.2 Looking inside models

One advantage of computer models is that it is possible to "look inside" the model at any stage during the simulation. Experimental work suffers from the problem that it is often problematic to observe certain aspects of a system whilst a process is taking place, and often methods of observing cellular phenomena can interfere with the phenomenon itself.

Such methods can be used for estimating parameters within real systems which are unobservable due to experimental constraints. In many situations a phenomenon is understood at the qualitative level, but the precise numerical value of rates, probabilities, et cetera is not known. An example of this is in protein/protein interactions in membranes, where the overall distribution of transmembrane proteins can often be observed using microscopy and tagging techniques (Hayes et al., 2003). Simulation can help to find feasible values for such parameters via the creation of a *parameterized space of models*, i.e. an abstract model is created which can be converted into a model of a hypothetical

situation via the specification of values for a set of free parameters.

The *parameters* of a model are the various choices which need to be made to realize a particular version of the model, e.g. the probability of certain events occurring, the affinity between proteins in a system, the initial distribution of entities in a system, functions describing the relationships between proteins when they interact, et cetera. These parameters can be considered to be variables which define a *parameter space*; any point in that space specifies a particular value for all the parameters and therefore provides a particular model. Some mathematical techniques can be used to analyse this space, e.g. breaking it down into regions with similar behaviour.

Experiments using the real system can then be carried out and data gathered. Then an optimization algorithm, such as hillclimbing (Reeves, 1993), tabu search (Glover and Laguna, 1993) or evolutionary algorithms (Mitchell, 1996), can be applied to search for a set or sets of parameters where the output from the situation matches the experimental observations. Whilst this provides only a *sufficient* set of parameters for realizing the experimentally observed phenomena, it provides a starting point for future experimental work.

An example of this is given in the work described in section 7 below and (Goldman et al., 2002, 2003). One of the motivations for developing this simulation is to be able to estimate probabilities of pairs of receptor proteins binding from large-scale time-series data about the size of clusters containing many such proteins.

3.3 Combining models and emergent phenomena

Another role of computer modelling is in synthesizing known facts about a system into a coherent model. For practical purposes experimental methods focus on a small part of a system at a time, and provide detailed results about particular parts of a system. However understanding the global behaviour of that system can be difficult. A computer simulation can allow these various results to be brought together into a single model. This can then be used, e.g. to check which of the individual parts of the system contribute to the various global behaviours observed, by removing or making changes to various components in the system and observing the effect on the behaviour which emerges as a result of the interactions.

Similarly, computational methods can be used to bring many items of data together into a coherent description. An example is given by Keedwell et al. (2002) and Keedwell and Narayanan (2003), where machine learning methods such as neural networks and genetic algorithms are used to synthesize many items of data about individual gene expression experiments into a network

summarizing the connections between the various proteins involved.

3.4 Modelling and sufficiency

The ability to simulate a particular observable phenomenon using some process does not demonstrate that that phenomenon must be produced by that process. Nonetheless it demonstrates that the process is *sufficient*. This demonstration of sufficiency can guide the research process in a number of ways. Firstly it can demonstrate that showing that the process in question is a valid option for a complete explanation for the phenomenon, which can suggest a line of experimental work leading towards demonstrating that the process is the explanation; similarly demonstrating that a process is insufficient (not something which can be done with simulation) can guide research towards the discovery of new components in the system. Also an understanding of the "minimal models" required to produce a particular phenomenon can be valuable because of evolutionary pressures towards simpler, more energy-efficient processes.

3.5 Bringing theory and experiment closer together

Another role of simulation is in helping to distinguish between various hypotheses to explain a particular set of experimental data. In some situations it is possible to fix many details of the system, however a number of different possibilities are available as to how the system works. Different models can be created to represent each of these possibilities, and simulations run based on these models. The results of these simulations can then provide a set of observational criteria for distinguishing between the various hypotheses. An example of this is given in (Whalley et al., 2002), where the computer is used to simulate two hypotheses for the behaviour of a certain protein, *viz.* whether it moves freely within the cell or remains bound to the cellular matrix. The simulation provides differing measures of cell lifespan according to which hypothesis is chosen. This suggests an experimental test to see which (if either) of the two hypotheses is satisfied.

The previous section provides specific examples of a general process through which simulation can be used. Traditionally theory has had to explain experimental results directly, by providing direct predictable phenomena for experiment to verify. Simulation can be used to "amplify" theory; in particular models can be built which are based on alternative theories, and the alternative outcomes provided by the simulation of these components compared with experimental data.

4 Examples of computer modelling of intracellular processes

A number of studies have been carried out applying computational techniques in the modelling and simulation of intracellular processes. One way to categorize these models is by the level of detail modelled by the system. Models at various levels of detail have a place in the scientific process, and it is naive to conclude that a more detailed model is "better" than a less detailed one. This section gives a number of examples of models and simulations of intracellular processes.

An example of the most detailed kind of model is given by the *E-CELL* system (Tomita et al., 1999). This is designed to reproduce the molecular interactions within the cell, for those cells where a complete DNA sequence is available, and for which a "complete" set of metabolic reaction rules can be written. The aim of a system such as *E-CELL* is to go beyond systems such as those described by Mendes (1993, 1997); Sauro and Fell (1991); Sauro (1993) which simulate single metabolic pathways, and to allow the computational study of the behaviour which emerges when these pathways are allowed to interact.

To specify a cell in the *E-CELL* system the user draws upon a library of existing objects, to describe both components within the system (an example of this is the *BindingSite* object) and the kinds of interactions between the objects which represent system components (e.g. the *MassActionReactor* object). These can be used "as is" or they can be specialized to a particular system by the use of inheritance (as discussed in section 6.4).

This provides a very powerful system for simulating the effects of interventions within the cell. However the cost of this is that a very detailed description of the cellular components and the reactions between these components is needed; for many such systems the details of the reactions in the cell, the genomic and proteomic data underlying these reactions and the pathways which arise from the interactions are currently only partially known.

A number of similar systems have been developed for more specialized applications. Two examples are the *Walk* system (Lamb, 1996; Lamb and Wischik, 1996) for the simulation of the G-protein cascade and the *StochSim* system (Morton-Firth and Bray, 1998) for understanding bacterial chemotaxis.

It is important to note that level of detail is a problem dependent choice, not a measure of model quality. The kinds of very detailed models surveyed in the previous section are very useful, if the detailed information about the system is available (e.g. for the *E-CELL* system a complete set of genomic and reaction data). However for many systems such data is not available, and indeed the role of the computer model is to help to fill in lack of information about the details of the system by working back from observed behaviour.

There would seem to be much scope for modelling which sits in the middle grounds between very detailed systems and broad conceptual models. Such models need to allow both for the incorporation of experimentally understood information where that information is available, but also for the incorporation of free parameters. The space of parameters defining this model-set can then be explored and matched against experimental data.

An example of this can be found in our own work on simulating the clustering behaviour of epidermal growth factor receptors in response to growth factor binding (Goldman et al., 2003). Existing experimental work with this system has determined which proteins interact; however these interaction affinities are unknown. We have developed a model which is parameterized by these affinities, and plan to estimate the rates by optimizing the model against observable phenomena from experimental work.

A final set of models are *qualitative* models. Many of these are based on the idea of grouping together parameter regions in which the behaviour of the system is (either exactly or broadly) the same. For example boolean models of gene expression (Narayanan et al., 2002), where a gene is regarded as being "active" or "inactive". Within the modelling of networks of biochemical reactions, methods which group together all qualitatively similar parameters and process the parameter set rather than an individual parameter have been devised (Reddy et al., 1996). Related work has also been carried out in the gene expression field (de Jong and Page, 2000; de Jong et al., 2001, 2002).

5 The modelling and simulation process

The creation of a computer simulation consists of a number of stages. There are many ways in which this breakdown can be achieved, depending on the type of problem being tackled, the preferences and experiences of the model-builders, and the level of formality required (e.g. in some industries a "paper-trail" is needed to understand which decisions have been made when in the development process). One problem with applying these methods is that many of them have been developed mostly with business applications in mind, and so the kinds of complexity that they are designed to deal with are perhaps different from the kinds of complexity dealt with in scientific simulation. However the features of the process are broadly similar in all domains.

A division can be made between a development phase where the software is created, and an application phase where it is used. For commercial software there is a clear distinction between these two phases and between the people involved in the phases. However in scientific software, where the developers of the software are sometimes the sole users, the distinction is less clear cut.

However a distinction can be made between the *modelling* phase where the real system is analysed and the software created, and the *simulation* phase where the software is applied and the results analysed.

The modelling phase can be broken down into three main activities. The first is *analysis* of the system at hand. In object-oriented methods this consists of identifying the various entities which play a role in achieving the phenomena of interest. At this stage large scale modelling decisions need to be made, e.g. outlining the scope of the system and its limitations. For example in a model of a signal transduction system will the model start from the assumption that ligands are being supplied to the cell at a particular rate, or will the ligand-producing system itself be incorporated into the model? Will the model incorporate the changing size/shape of the cell as it grows and divides, or will this be kept constant? How much detail is it necessary to include in the model: is it important to keep track of individual molecules, or are just the quantities of each required?

The next stage is a *design* phase, in which computational structures representing the entities identified in the analysis phase, and the relationships between these entities, are designed. In the object-oriented method which is discussed further below, this consists of identifying the information which is required to represent an abstraction of each entity, and a description of the possible actions which such an entity is capable of carrying out. At this stage the designs are written descriptions in natural language (e.g. English), supported perhaps by diagrams (particularly to explain the relationship between parts of the design).

The final part of the modelling stage is the *implementation*. This is where the design is formalized into a computer language such as *Java* (Arnold and Gosling, 1997) or C++ (Stroustrup, 1991). This consists of taking each part of the design, deciding how the information required to specify that part of the design is to be represented on the computer, and how that information changes as each of the possible actions is carried out. Some examples of what a computer language model looks like are given in table 1 and figure 7.

Once this process has been carried out the model is ready to be applied. However it is likely that the various stages above will need to be revisited. For example the application of the model may point to an aspect of the system which was not incorporated into the original model, or the scope of the system may need to expand to cope with additional research questions thrown up by the use of the model. This requires working back to the analysis stage, deciding how these new features are going to be incorporated into the original design (which may involve changing the original design (Fowler, 1999)), and then implementing these changes. Alternatively the change to be made may occur later on in the process. For example some important part of the simulation may

take too much time to carry out its calculations; in this case it is not necessary to reanalyse the original system, but to replace the existing implementation of that part of the simulation with a more efficient one (if one can be devised).

A transitional stage between the development of the model and its application is *testing*. A number of basic tests need to be carried out to check whether the individual parts of the simulation work in isolation. Typically this is carried out by writing an additional program (sometimes called a *test harness* by analogy with the structures built to test engineering components) which supplies that part of the simulation with a sequence of input data for which the expected response is known, then checking that the state of that part of the simulation after this input is as expected. For example consider testing a part of a simulation which responds for collisions between particles. In such a case the test harness might create pairs of particles, some which are on a collision course and some of which are not, and check that (1) those which are expected to collide are flagged as having collided, and *vice versa*, and (2) that the final trajectory of the particles is as would be expected from the mechanics of a two-particle collision.

However this testing of individual components is only the first part of the testing strategy. In addition the components must be tested together on some predictable scenarios to provide a check that the behaviour which emerges from the interactions between the components is as expected. For example in our work on receptor clustering (Goldman et al., 2002, 2003) the diffusion behaviour emerging from a large number of particle interactions was measured and compared against theoretical predictions from brownian motion theory (Berg, 1993).

Clearly all such testing is limited in scope. The main purpose of testing is to enable software developers to discover errors in the implementation of the design, rather than to show that the system works correctly. More formal structures for verifying properties of systems are available; however, formulating a precise specification of the desired system can be a difficult task in its own right. An analogy can be drawn here with the process in wet experiments of checking that the experimental processes being used do not interfere with the processes which the experiment is designed to study.

Once the system has been tested it can be applied to simulating various experimental situations. The process is not dissimilar to laboratory experiment: typically the simulation is run many times, with differing initial conditions and random fluctuations in the background conditions to ensure that the observed phenomena are stable and are not merely the result of a particular finely-chosen parameter set. These *virtual experiments* are "observed" by storing various values as the simulation progresses: this can involve both looking inside the model to measure certain characteristics, and making "observations"

(i.e. doing calculations) of emergent features of the model.

At present most simulations are carried out in this way. However there is considerable scope for exploiting the features of the simulations *qua* programs, which have not been heavily used in research in this area to date. One example is potentially being able to reverse the process, and work backwards from observed final results to the initial conditions which led to those results. In particular recent work on "backward analysis" of programs (King and Lu, 2002) has been concerned with calculating the space of inputs to a program given a final state. Another aspect of treating models as programs to be analysed rather than just executed on particular data sets is the application of qualitative models. A number of techniques (e.g. abstract interpretation of programs (Nielson et al., 1999) and qualitative reasoning systems (Lee, 1999; Kuipers, 1994)) allow a program to process a whole set of inputs rather than a single input. This allows users to ask more general questions like "how will the model respond when a parameter is above a particular critical value" rather than simulating for a small number of fixed values. There are limitations on what kind of information can be obtained using such methods; nonetheless, there is considerable scope for the application of such ideas. This kind of work is commonly found in differential-equation type models; once such a model has been created it is sometimes possible to solve simple versions analytically, techniques can be applied to analyse the qualitative behaviour of the system, and numerical analysis can be used to ascertain the behaviour of the system under a particular choice of parameters and initial conditions. Program analysis techniques have the potential to expand this range of analysis to a broader range of models.

6 An introduction to object-oriented methods

The aim of this section is to describe a method of software creation which is known as the *object-oriented* method. This provides ways of analysing problems/systems, designing solutions to those problems and models of those systems, and implementing those solutions/models on the computer using an object-oriented programming language (C++ and Java are heavily-used examples).

6.1 Implementation complexity and problem decomposition

A different kind of complexity to the types discussed in section 2 is the complexity created by implementing systems on the computer. Simple systems can be easily understood and held in the mind of the programmer whilst

the program is being created. However problems do not need to be too sophisticated before a single person cannot hold all aspects of the problem in their mind simultaneously. This necessitates a breakdown of the problem into subcomponents, each small enough to be understood and implemented by a single programmer in a reasonable amount of time. Such subcomponents can be worked on sequentially by a single programmer, or they can be farmed out between a number (in some cases, several hundred) of programmers (Booch, 1994).

This *decomposition* of the system should spilt the problem down into a small number of manageable units which can be worked on without knowing the fine detail about the other units. Ideally the only information a programmer should need to know about the other units in the system is how to pass information to them, and what sort of information it might receive from them.

There are two main ways in which this problem decomposition can be carried out. The first (functional decomposition) consists of looking at the process being simulated and considering the various processes which occur as the process happens, and the effect that these processes have on the data representing the aspects in the model. The second form of problem decomposition is *object-oriented decomposition*. In this the process being simulated is broken down by considering what entities within the model are important to the process, and creating computer models of the *state* and *behaviour* of those entities.

In recent years a mixture of theoretical arguments and practical experience has favoured the object-oriented decomposition as the method of problem breakdown which is most suited to dealing with the various types of complexity outlined above. An extended exegesis of this can be found in (Booch, 1994).

6.2 Classes and objects

The basic process of object-oriented modelling consists firstly of identifying the types of entity which are involved in a particular system. For example, in modelling a cellular system the following might be examples of such types: the cell itself, the receptors on the cell surface (both in general and the specific receptor types), the proteins within the cell, et cetera. These are called *classes*, and are defined by a *state* (the type of information required to give a "snapshot" of an example of that entity at a particular moment; each piece of information is called an *attribute*) and potential *behaviours* (the types of actions which are permitted for entities of that type, and possibly some constraints and conditions under which these behaviours will occur) which entities of that type can carry out. An individual behaviour is called a *method*.

As an example consider a particular type of protein within a cell which is

specified by a class as follows. The state at a particular point of time is given by a *position* within the cell, a current *velocity* and a specification of whether the protein is phosphorylated or not. This could be represented by eight attributes: three decimal numbers to represent position, three to represent a direction vector, a true/false value to say whether or not it is phosphorylated, and a reference to the region in which it is found. The protein exhibits a number of behaviours, each of which has a method associated with it to tell the computer how to change the information within the cell when that behaviour occurs: e.g. a *move* method which changes its position by its current velocity, and a *phosphorylate* method which changes the state of the true/false attribute *phosphorylated* to true.

These classes form templates for the creation of *objects* in the simulation. These objects are particular realizations of the classes. As an example a particular kind of molecule might be a class; within a simulation it would be possible to create thousands of *instances* of that molecule, each of which is described by the same *type* of information (position, velocity, . . .) but where the particular *value* of those pieces of information differs from molecule-to-molecule (these are referred to as the *attributes* of the class). It can be seen from this discussion that the creation of an object of a particular class consists of the specification of an initial value for each of the attributes in the class. In addition to this the object is given a particular unique (in its context) name. This act of creating an object of a particular class is achieved by a specific method within the class called a *constructor*, which is always the first method to be called when a new object of that class is created.

Once these classes have been created, a model can be built by creating an appropriate number of objects of appropriate types to match the system in question, and manipulating these objects via their behaviours. These behaviours can be triggered by a central controlling program, which dictates the order in which the behaviours are carried out. However good practice delegates much of this behaviour to the objects themselves. For example instead of a central program calculating the probability of two proteins binding when they are near to each other in the cell, this is carried out by methods within the objects representing the proteins themselves. Under this principle the main control program is relegated to a simple role of setting up the objects in the system, setting in chain the initial interactions between those objects, and (possibly) coordinating the various objects by providing a notion of time in the simulation. An example of such a simulation structure is given in section 7 below.

To create an object oriented simulation the programmer needs to analyse the real world situation then design a set of classes and relationships between those classes which reflect the situation being studied.

The first stage of the analysis is to identify the classes which the simulation will need to take into account. A number of techniques can be used to assist with this. One technique is to find one class and then work outwards from that by considering which other kinds of things in the system objects of that class need to interact with. So for example the class *receptor* might be identified. Then working out from this we observe that receptors receive information in the form of ligand and pass on information to second messenger proteins, thus suggesting two other classes (these can then be further refined, by the use of inheritance, into specific types of ligands and second messengers). Also the receptor needs a membrane in which to exist; this provides another class.

Another technique is to take texts about the process in question, and to identify the nouns and noun-phrases in that text. Here is an example, which could be the starting point for the development of a (fairly abstract) model of gene expression.

> Gene regulatory proteins must recognise specific nucleotide sequences embedded within this structure. [...] It is now clear, however, that the outside of the double helix is studded with DNA sequence information that the gene regulatory proteins can recognize without having to open up the double helix. The edge of each base pair is exposed at the surface of the double helix, presenting a distinctive pattern of hydrogen bond donors, hydrogen bond acceptors, and hydrophobic patches for proteins to recognize (Alberts et al., 2002).

The underlined phrases represent an initial list of potential classes. The next stage is to refine this list. One aspect of this is removing or modifying words which are at the wrong level of description for the problem at hand. In the above example is it unlikely that the "double helix" will need to be represented explicitly, all that is needed is the "DNA sequence information". Thus the two can be conflated into a single class. Also we note that "proteins" in the last sentence is being used as shorthand for "gene regulatory proteins", so we can conflate the two. Some details will also be left out. For example there is unlikely to be any need to explicitly represent geometrical features of the DNA such as the "edge" and "surface". Another part of this process is to recognise that some aspects of the description form parts or specialized variants of others. In the above example the "distinctive pattern" is made up of three components: "hydrogen bond donors", "hydrogen bond acceptors", and "hydrophobic patches".

The next stage in the analysis process is to map out the relationships between the identified classes. Again this can be supported by the analysis of relevant texts, looking this time for verbs which link concepts together. For example in the text above the last sentence discusses the idea that protein objects will need to "recognize" the distinctive patterns in the DNA sequence. This network of interactions can often be best notated using a diagram. In recent years a standard known as the Unified Modelling Language (UML) has been designed to represent the various kinds of relationships within object oriented systems (Fowler, 1997; Stevens, 2000). The most commonly used diagram from this notation is the *class diagram*, which shows classes and their interrelationships. An example of such a diagram for the system discussed above is given in figure 4 and a more detailed example in figure 6. The notation at the end of each line indicates how many objects of each class can exist. For example a given feature set can contain from 0 to any number (0..$*$) of hydrophobic patches.

Once this set of classes and their interrelationships has been created, the design phase of the process begins, which is concerned with converting the ideas from their real world domain to a computational description. The key aspect of this is deciding which information about the real world objects need to be stored in their class description, and how these are to be structured so that the computer can act on them in an efficient fashion. The information required will depend on the use to which the classes are being put. For example in the model of gene expression discussed in above, it is unlikely that the detailed three-dimensional structure of the proteins will need to be represented within the *gene expression protein* class; in a protein class being used for studies of the detailed molecular dynamics of the binding between two proteins (Alonso et al., 2001), this information is essential.

6.4 Relationships between classes

Many classes have some similarity. For example the computer representation of any protein in the cell will need a certain set of basic information and behaviours, such as its position. However some proteins will have specific features, for example a receptor protein will have the scope for being liganded or not, a prion protein can exist in both a normal and abnormal configuration.

Clearly it does not make sense to rewrite the entire class each time a new protein type is introduced. Even if the original were copied and modified, there are potential problems; e.g. if an error is discovered in the implementation of one of the basic behaviours, or a more efficient way of implementing that behaviour is discovered, then this behaviour needs to be modified in *all* places where it occurs. One way to do this is via careful record-keeping; however this

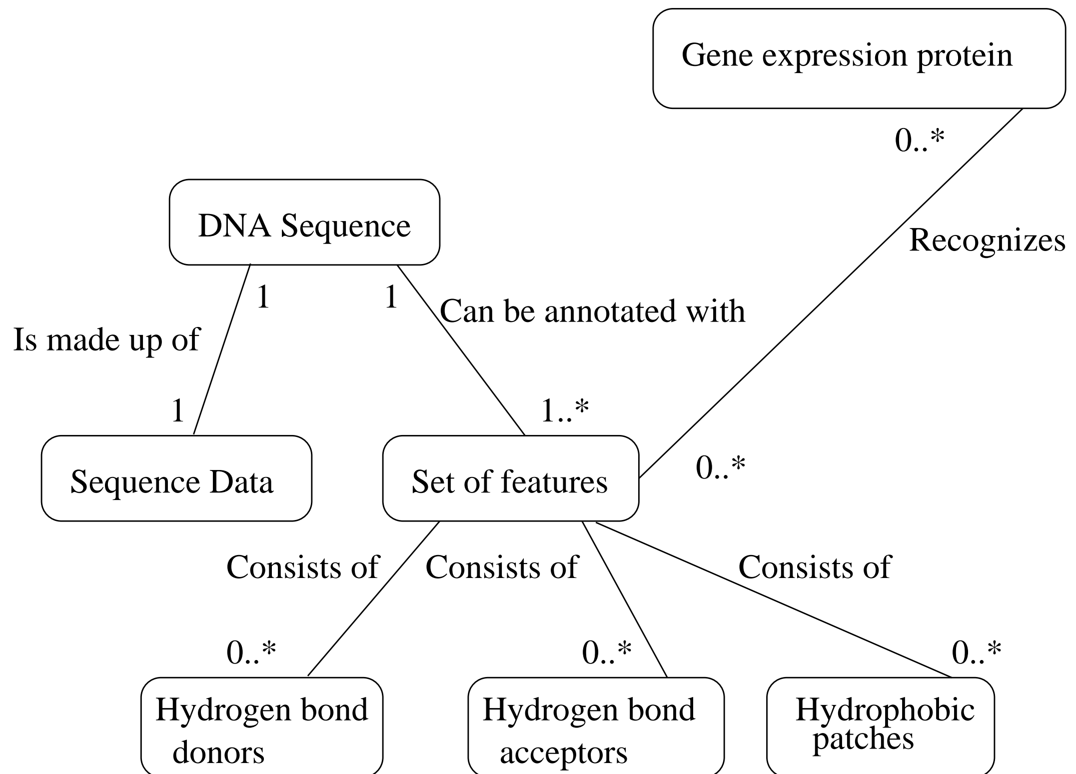


Fig. 4. An example of a UML class diagram.

process is automated in object-oriented computer languages by the notion of a *class hierarchy.*

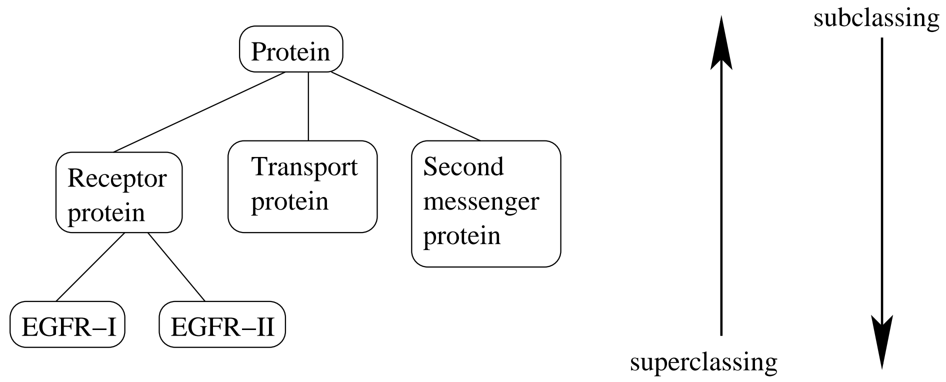
The basic idea behind this is that some classes can *inherit* their behaviour from others. The class doing the inheriting is called a *subclass*, and the one it is inheriting from called the *superclass* (note that these labels are relative; the same class can play the role of a subclass relative to one class and a superclass relative to another). The core idea of inheritance is that a subclass has all of the attributes and methods of its superclass (and, recursively, the superclass of that superclass, . . .), unless specifically overridden. So for example a class might be created to represent the general concept of protein. "Receptor protein" is a subclass of this (it has all the characteristics of a protein, but in addition it has certain aspects of state and behaviour which are specific to its role as a receptor, e.g. the ability to bind ligand). Carrying on further through the hierarchy, "human type III epidermal growth factor receptor" is an example of a subclass of receptor protein; it *is* a receptor protein (and, recursively, a protein), but it has certain characteristics which are not shared by other receptor proteins, such as a certain set of ligands to which it is allowed to bind.

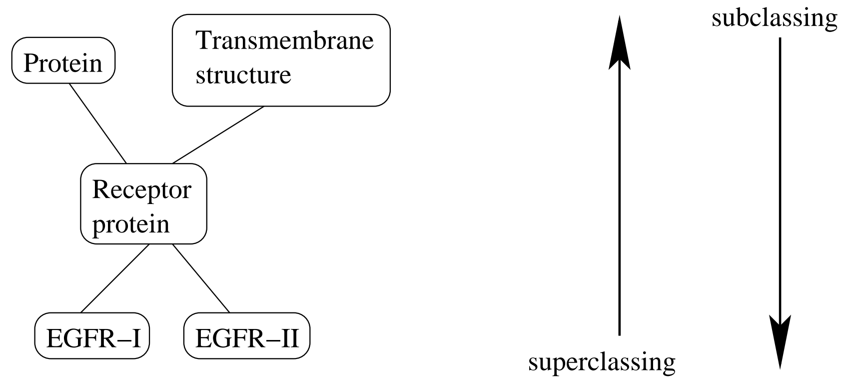
Different OO computer languages allow differing levels of sophistication in this inheritance hierarchy. For example C++ allows *multiple inheritance*, e.g. a receptor protein could inherit characteristics both from the protein class and from a class of "transmembrane objects". However this provides a number of technical problems (in particular relating to the same piece of real-world information accidentally being represented twice or more within the same class), so other languages are more restrictive. For example in the *Java* language classes are only allowed to inherit from one class, but they are allowed to bring in a large number of *abilities* (called *interfaces*). In a biochemical context such abilities might be a feature of some molecules such as being able to be methylated or phosphorylated. These can be freely added in a consistent fashion to the classes using the interface mechanism, whilst keeping all of the information about e.g. methylation in a single place in the program.

These various forms of hierarchy are summarized in figure 5.

6.5 Implementing object orientation.

Once this design has been created, the final stage in producing a model is to *implement* the model in a computer language. In order to do this the programmer takes each of the classes described in the design and translates the design into a computer language. The attributes of each class become computer-friendly representations of the information required to describe the





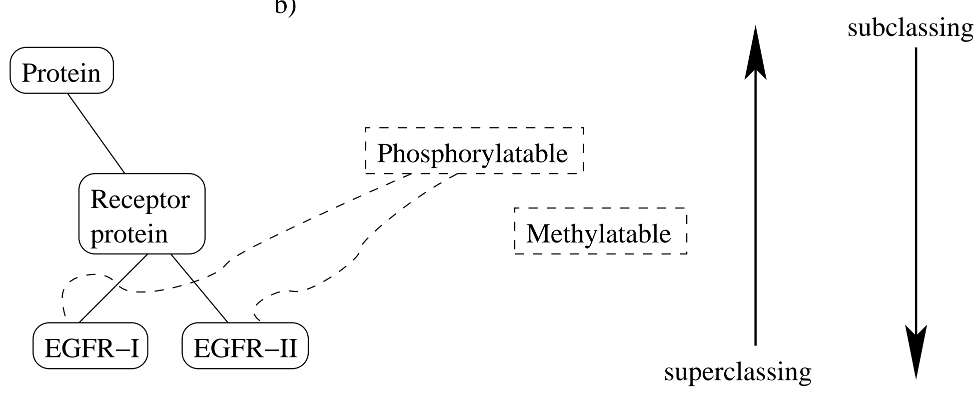


Fig. 5. Inheritance structures. *(a) a basic example of single inheritance (b) multiple inheritance, as found in the C++ language (c) interfaces, as found in the Java language.*

attributes, whilst the methods become descriptions of how that information changes when that particular piece of behaviour is carried out.

6.6 Design and implementation techniques: patterns, structures, algorithms

An experienced programmer does not tackle each new design and implementation from scratch. They approach new problems with a repertoire of well-known techniques which can be applied to a number of situations. Rather than learning these through experience, a number of them have been semi-formalized so that they can be learned as part of learning to create software.

At the analysis and design stages of software creation, *patterns* (Gamma et al., 1994) provide abstractions of common situations which are found when translating real-world systems into computer models. Two examples of such patterns are the notion of a *Prototype*, where a template example object of a class is created and new objects generated by copying and modifying that class, and an *Iterator*, where the items in a collection are retrieved and acted upon sequentially. These are two examples of patterns which are used at the design stage of creating a simulation(Gamma et al., 1994) Patterns can also be used at the analysis stage (Fowler, 1996). The aim of all such applications is to provide standardized ways of going about common processes, abstract enough that they can be applied to many different problems, but with enough detail so that they provide a useful guide as to how to tackle problems of that type.

Much work on patterns is about programming in general; however some (particularly the analysis patterns) are more specific to application domains. Given that (teaching of) programming is dominated by programming for business applications, some of these are less relevant for scientific applications. Providing sets of patterns appropriate for scientific programming is an interesting future challenge.

At the implementation level *data structures* and standard *algorithms* play a similar role (Knuth, 1981, 1997, 1998; Skiena, 1997). Examples of common data structures are lists, tables, tree-like structures. Examples of algorithms are searching for a particular value, putting a list into order. These appear in many different problem types, so a small effort in improving how these are implemented can redound to an improvement in many different application areas. Modern computer languages (Java being a particularly notable example) have efficient implementations of many such structures as a part of the language.

6.7 *Summary*

Table 2 summarizes the core concepts explained in this section.

6.8 *Object-oriented methods for science*

One potential problem with the application of object-oriented methods is that they have been designed principally with business applications in mind rather than scientific applications. Therefore it is interesting (and an important and neglected research area) to consider ways in which the methods described above might be adapted specifically for the creation of scientific software. In particular OO methods have been designed with the kind of complexity in mind which arises from many *people* working in a complex information environment, with conflicting needs and differing ontologies (i.e. different uses of language and different ways of classifying the same information) about the information they share.

One example of where this difference can be found is in the calling of methods (i.e. the performance of the behaviour described in that method). In OO languages such as Java, there is little ability to constrain when a particular method can be activated. This is natural in a business context; e.g. in a system concerned with managing airline bookings, the method *bookFlight* can be called freely, whenever someone wants to attempt a flight booking. However in a molecular simulation, the method “moveMolecule” is constrained to occur once in each timestep. At present the responsibility to ensure that this happens once-and-only-once per timestep is the responsibility of the programmer at the implementation stage, and if another programmer working to improve the program at a later stage violates this constraint, it is not automatically flagged up when the program is compiled or run. Adding in the ability to directly enforce such constraints between objects of two different classes would be a useful further development of the OO model with particular relevance for scientific modelling.

7 Simulating intracellular processes

The previous section has focused on generic issues of object-oriented programming, with examples of details being given from cellular systems. In this section more details about an overall framework for simulating intracellular phenomena will be given.

Most simulations of cells can be split down into three stages. The first of these is an *initialization* stage in which the main object to be used in the simulation are created. For example a object will be created to represent the cell itself, and examples of the various kinds of proteins which will be important in that cell.

Typically object-oriented models are used when there is a reason to have an independent representation for each of the interacting objects in the process being studied. A common reason for this in cellular biology is because of spatial heterogeneity within the cell, e.g. structures such as *signalling complexes* (Bray, 1998) or *lipid rafts* (Carpenter, 2000). By contrast some processes can be well represented simply by keeping track of the relative concentrations of objects in the simulation, under assumptions that they are in a well-mixed spatially homogeneous environment (examples of simulations of this type are the SCAMP (Sauro and Fell, 1991; Sauro, 1993) and *Gepasi* (Mendes, 1993, 1997) systems).

The main, central part of the program is concerned with running the simulation. Most simulations are based around the idea of discrete *timesteps*, i.e. very short simulated time intervals. An alternative is to use event-driven simulation, where instead of a sequence to timesteps driving the events in the system, the program calculates ahead to the next event and calculates the state of the system at the event and the consequences of the event on that state (Sigurgeirsson et al., 2001). For the purposes of this paper the idea of timesteps is used.

In order to construct the model the programmer needs to specify what occurs in each timestep, and how the program should deal with any events which occur within that timestep. Usually the detailed calculations will be encapsulated within the objects involved, the sequence of events within the timestep being concerned mainly with triggering methods within the objects as appropriate.

For example a timestep within some model of proteins within a cell which can associate to form pairs and where these pairs can dissociate might consist of the following steps:

- Calculate where the proteins will move during the current timestep.
- Check which of those proteins will come close enough so that they might associate during that timestep.
- For those which do come close enough, check whether they will associate by making a randomized choice based on the affinity of interaction.
- For each pair that bind, replace the two protein objects with a single object representing the bound pair.
- For each pair which do not bind, calculate their new trajectories based on

their collision.
- Check whether each current pair will dissociate, by making a randomized choice based on the probability of dissociation.
- Store information about the current state of the system.

This process is repeated many times, e.g. for a fixed number of cycles or until some equilibrium condition is met.

The third and final stage is summarizing and processing the data gathered. This can be in the form of summary statistics, graphs and charts, animations of the process, statistical hypothesis testing, et cetera. These different outputs play differing roles: a formal statistical comparison with experimental data may be needed to provide evidence for some hypothesis; by contrast, a graphical animation of the system might be used to provide an informal comparison with what the scientist is accustomed to viewing via microscopy.

As an example of this process consider the system we have developed to model the clustering behaviour of epidermal growth factor receptors on the surface of cells. Further details of this can be found in Goldman et al. (2002, 2003).

The system being modelled consists of receptors which move around freely (under the influence of brownian motion) on the surface of the cell. In their unliganded state they have few interactions; occasionally pairs will associate, however these pairs are unstable and dissociate soon after formation. Once they have bound ligand, however, the association between pairs becomes very strong, and the pairs themselves can associate to form larger clusters. The system is explained in detail in (Salomon and Gullick, 2001; Schlessinger, 2000; Yarden and Sliwkowski, 2001; Carpenter, 2000; Gullick, 2001). The number and size of these clusters can be measured experimentally by tagging the receptors (or associated second-messenger proteins) with fluorescent proteins and tracking the movements by light microscopy (Gillham et al., 1999; Monks et al., 1998; Hayes et al., 2003). The size of these clusters is important in determining the strength of the downstream signal.

The analysis of the system identified the following classes:

- Receptor molecules
- The cell surface
- Multimers, consisting of a number of molecules
- A table giving the probability of association between receptors and multimers.

As the system way developed a number of additional classes were added, not to represent entities found in the system itself, but to represent aspects of the user interface to the program and the data which needs to be gathered as the program runs:

- The accumulated data about cluster size and structure with time
- A graphical user interface to the system
- A system to produce graphs of the data

The relationship between these is shown in a class diagram in figure 6. In particular this shows how many objects of each class are allowed/obliged to be associated with each other class, and the nature of these associations. So for example each `CellSurface` object can be associated with one or more receptor `Molecules` ($1 \ldots *$), whereas each receptor `Molecule` is associated with one and only one `CellSurface`.

During the design phase of the process a number of decisions were made about the level of detail required in the simulation. For example it was considered unnecessary to model any of the cellular molecules which are not involved in the interactions, but instead to implement a brownian motion algorithm to represent interactions between receptors and other transmembrane and near-membrane molecules. Another example of a decision taken at this stage was to model the association and disassociation between molecules by a simple probabilistic process rather than by e.g. making a more accurate model of the three-dimensional shape of the various components. These decisions reflect a choice of a certain level of detail in modelling, appropriate to (1) the level of knowledge we have about the system (2) the desired level of abstraction the model and (3) the amount of computation time available.

Much of the above was time-consuming to implement but comparatively straightforward. Figure 7 gives the first few lines of the Java code for the `Molecule` class, demonstrating how the class is structured and introducing the attributes and some examples of methods.

An important challenge in implementation was getting the system to detect collisions efficiently. Checking each potential pair of receptors for collision rapidly becomes impractical, even on fast computers, as the number of receptors increases. As a result various methods of breaking down the cell surface into manageable substructures were experimented with; see Goldman et al. (2003) and Johnson and Whalley (2002) for details.

Currently this system is being applied in a number of ways. Firstly it is being used to investigate the contribution of the various parameters in the system to the overall behaviour, e.g. by looking at the behaviour of the system for different values of association and dissociation constants. The aim of this is to investigate which of these processes have most influence on the global phenomena observed. Ongoing work is focused on using the system to estimate parameters, in particular using optimization techniques to adjust the parameters in the model so that time series of cluster sizes match respective data from real experiments, so that feasible values of experimentally-inaccessible

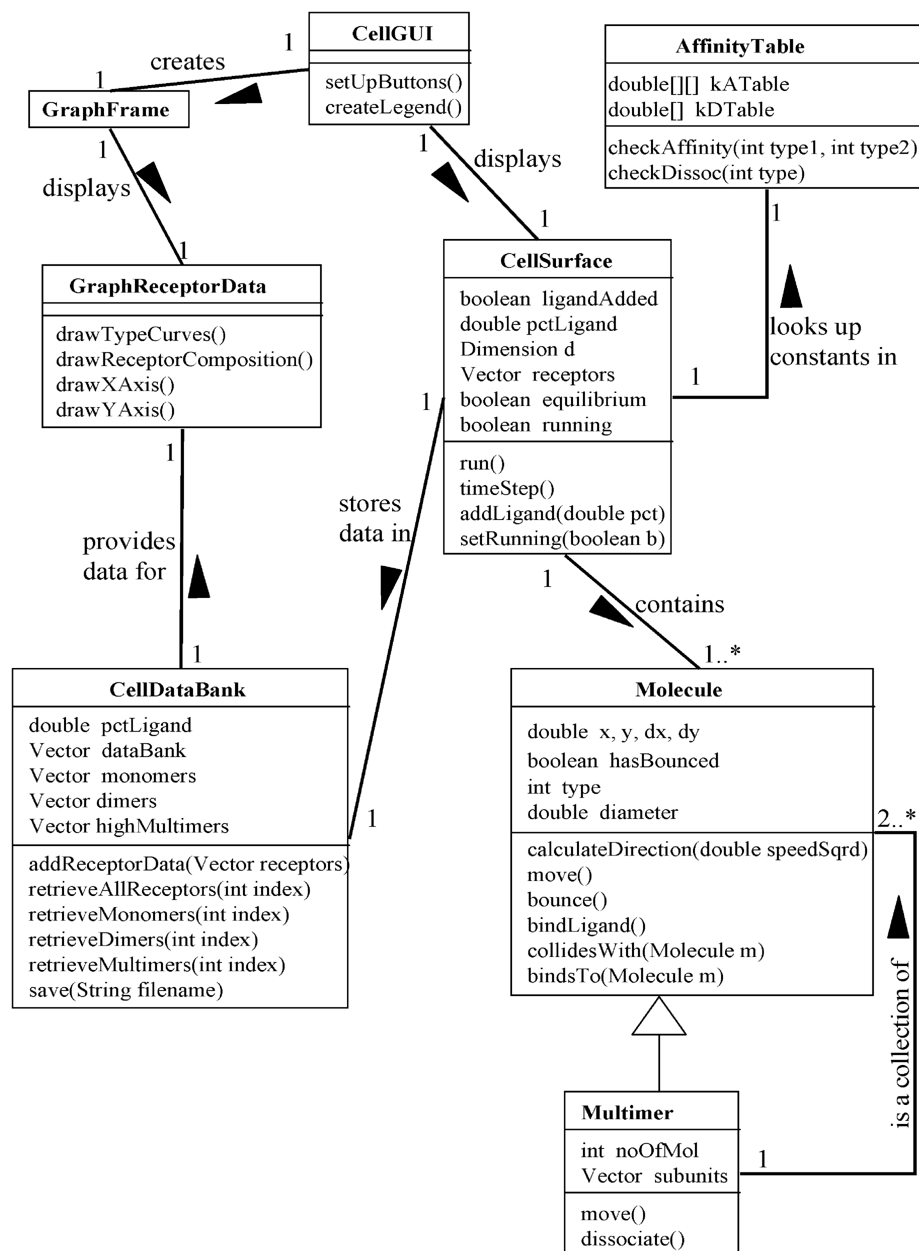


Fig. 6. A UML diagram summarizing the relationships between the classes in the receptor clustering model.

```java
import java.awt.*;
import java.util.*;
import java.awt.geom.*;
import java.io.*;

/** A class to represent a generic cell surface receptor monomer
 */
public class Molecule implements Serializable, Cloneable  {

    // attributes

    protected double x, y, dx, dy;
    protected int type;
    protected double diameter;
    protected transient Grid grid;
    protected transient Vector gridPoints = new Vector();

    // constants

    /** A constant for the diameter of the circle representing
        a receptor monomer */
    public static final double DIAM = 1.0;

     public Molecule() {
    }

    /** Constructor to create Molecule in a random position within
     *  the area whose size is given by the Dimension parameter.
     *  @param d Dimension within which Molecules position is set
     */
    public Molecule(Dimension d)    {
        x = Math.random()*d.width;
        y = Math.random()*d.height;
        calculate_direction(CellSurface.SPEED);
        type = AffinityTable.MONOMER_NOLIG;
        diameter = DIAM;
    }

    /** Constructor for Molecule whose starting position is known
     */
    public Molecule(double x, double y, double dx, double dy) {
     this.x = x;
     this.y = y;
     this.dx = dx;
     this.dy = dy;
     type = AffinityTable.MONOMER_NOLIG;
     diameter = DIAM;
    }

    /** Moves the Molecule in a random direction to a new position.
     *  The distance travelled is constant based on the size of the Molecule.
     */
    public void move()  {

        calculate_direction(CellSurface.SPEED);

        x+=dx;
        y+=dy;

        updateGridPoints();

    }

    /** Deals with a non-productive collision with another Molecule
     *  (i.e. when no binding occurs).
     *  @param m the colliding Molecule
     */
    public void bouncesOff(Molecule m) {
        // this should reset the position of the Molecule
        // so that it is just outside the limits
        // of the one it's bouncing off

        Molecule bouncer = this;
        if (bouncer instanceof Multimer) {
            bouncer = (Multimer)bouncer;
        }

        // make a triangle from 3 points, the current location,
        // the previous location, and the location of m.

        double x_now = bouncer.getX();
        double y_now = bouncer.getY();
        double x_m = m.getX();
        double y_m = m.getY();
        bouncer.reverse();

        double x_then = bouncer.getX();
        double y_then = bouncer.getY();

        double a = Math.sqrt((x_now - x_then)*(x_now - x_then) +
          (y_now - y_then)*(y_now - y_then));
    ...
```


Fig. 7. The beginning of the Java language definition of the `Molecule` class.

parameters can be found.

8 Conclusions and prospects

This paper has given an outline of object oriented modelling for modelling intracellular processes. To finish a number of suggestions for further reading are given.

8.1 On object-oriented methods in general

- Grady Booch, *Object Oriented Analysis and Design.* A fairly old book now, but strong on justifying why the OO methodology is important. A good discussion of complexity in science and in human endeavour, and an attempt to classify different types of complexity. A new edition has been promised for several years now (Booch, 1994).
- Iain Craig, *The Interpretation of Object-Oriented Programming Languages.* Good for details on why OO languages are structured as they are. Rather technical (Craig, 2002).

8.2 On OO programming languages

- *Java* is a good all-round programming language suited to beginners, though still not trivial to learn. There are many good books on the *Java* language, e.g. (Barnes, 2000; Winder and Roberts, 2000; Arnold and Gosling, 1997).
- C++ is heavily used in commercial applications. The standard reference is (Stroustrup, 1991), but this is not an introductory book for non-programmers.
- *Python* is an increasingly popular object-oriented language which is both easy to learn and good for beginners. See `http://www.python.org/` and (Lutz and Ascher, 1999).

8.3 On notations to support OO design

- Martin Fowler, *UML Distilled.* Short and to the point with reasonable examples (though, as with all the books on this topic, the examples are uniformly business-related rather than science related (Fowler, 1997).
- Perdita Stevens, *Using UML.* Lots of detailed examples, again drawn from business applications and similar "people processes" (Stevens, 2000).

There are a small number of case studies and analyses which specifically address OO methods in science: some examples are (Norton et al., 1996, 1995; Nemirovsky, 1994).

9 Acknowledgements

Many thanks to Dennis Bray and Jacqueline Whalley for discussions on this subject.

References

Alberts, B., Johnson, A., Lewis, J., Raff, M., Roberts, K., Walter, P., 2002. Molecular Biology of the Cell, 4th Edition. Garland Science.

Alonso, D. O., DeArmond, S. J., Cohen, F. E., Daggett, V., 2001. Mapping the early steps in the pH-induced conformational conversion of the prion protein. Proceedings of the National Academy of Sciences 98 (5), 2985–2989.

Alonso, D. O. V., An, C., Daggett, V., 2002. Simulations of biomolecules: characterization of the early steps in the ph-induced conformational conversion of the hamster, bovine and human forms of the prion protein. Philosophical Transactions of the Royal Society of London A 360 (1795), 1165–1178.

Arnold, J., Gosling, K., 1997. The Java Programming Language. Addison-Wesley, second edition.

Barnes, D., 2000. Object-Oriented Programming with Java. Prentice Hall.

Berg, H. C., 1993. Random Walks in Biology. Princeton University Press, expanded second edition.

Booch, G., 1994. Object-oriented design with applications, 2nd Edition. Benjamin/Cummings.

Bray, D., 1998. Signalling complexes: Biophysical constraints on intracellular communication. Annual Review of Biophysics and Biomolecular Structure 27, 59–75.

Bray, D., 2002. Bacterial chemotaxis and the question of gain. Proceedings of the National Academy of Sciences of the United States of America 99 (1), 7–9.

Bray, D., Levin, M. D., Morton-Firth, C. J., 1998. Receptor clustering as a cellular mechanism to control sensitivity. Nature 393, 85–88.

Carpenter, G., 2000. The EGF receptor: a nexus for trafficking and signalling. Bioessays 22 (8), 697–707.

Colton, S., Bundy, A., Walsh, T., 2000a. Automatic invention of integer sequences. In: Proceedings of AAAI-2000.

Colton, S., Bundy, A., Walsh, T., 2000b. On the notion of interestingness in automated mathematical discovery. International Journal of Human Computer Studies 53 (3), 351–375.

Craig, I., 2002. The Interpretation of Object-Oriented Programming Languages, 2nd Edition. Springer.

de Jong, H., Geiselmann, J., Batt, G., Hernandez, C., Page, M., 2002. Qualitative simulation of the initiation of sporulation in *B. subtilis*. Tech. Rep. 4407, INRIA.

de Jong, H., Page, M., 2000. Qualitative simulation of large and complex genetic regulatory systems. In: Horn, W. (Ed.), Proceedings of the 14th European Conference on Artificial Intelligence. IOS Press, pp. 141–145.

de Jong, H., Page, M., Hernandez, C., Geiselmann, J., 2001. Qualitative simulation of genetic regulatory networks: Method and application. In: Nebel, B. (Ed.), Proceedings of the 17th International Joint Conference on Artificial Intelligence.

Fowler, M., 1996. Analysis Patterns. Addison Wesley.

Fowler, M., 1997. UML Distilled. Addison-Wesley.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley.

Freeland, S., Hurst, L., 1998a. The genetic code is one in a million. Journal of Molecular Evolution 47 (3), 238–248.

Freeland, S., Hurst, L., 1998b. Load minimization of the genetic code: history does not explain the pattern. Proceedings of the Royal Society of London B 265, 2111–2119.

Freeland, S., Knight, R., Landweber, L., Hurst, L., 2000. Early fixation of an optimal genetic code. Molecular Biology and Evolution 17, 511–518.

Freitas, A. A., 1999. On rule interestingness measures. Knowledge-Based Systems Journal 12 (5–6), 309–315.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. Design Patterns. Addison-Wesley.

Gillham, H., Golding, M. C., Pepperkok, R., Gullick, W. J., 1999. Intracellular movement of green fluorescent protein-tagged phosphatidylinositol 3-kinase in response to growth factor receptor signalling. Journal of Cell Biology 146 (4), 869–880.

Glover, F., Laguna, M., 1993. Tabu search. In: Reeves (1993), pp. 70–150.

Goldman, J. P., Gullick, W. J., Bray, D., Johnson, C. G., 2002. Individual-based simulation of the clustering behaviour of epidermal growth factor receptors. In: Lamont, G. (Ed.), 2002 ACM Symposium on Applied Computing. ACM Press.

Goldman, J. P., Gullick, W. J., Johnson, C. G., 2003. Individual-based simulation of the clustering behaviour of epidermal growth factor receptors. Scientific Programming In press.

Gullick, W., 2001. The type 1 growth factor receptors and their ligands con-

sidered as a complex system. Endocrine-Related Cancer 8, 75–82.
Hayes, N., Howard-Cofield, E., Gullick, W., 2003. Green fluorescent protein as a tool to study epidermal growth factor receptor function. Cancer Letters In press.
Hornos, J., Hornos, Y., 1993. Algebraic model for the evolution of the genetic code. Physical Review Letters 71 (26), 4401–4404.
Johnson, C. G., Whalley, J. L., 2002. Detecting collisions in sets of moving particles: a survey and some experiments. Tech. Rep. 8-02, University of Kent.
Keedwell, E., Narayanan, A., 2003. Genetic algorithms for gene expression analysis. In: Raidl, G., Cagnoni, S., Cardalda, J. R., Corne, D., Gottlieb, J., Guillot, A., Hart, E., Johnson, C., Marchiori, E., Meyer, J.-A., Middendorf, M. (Eds.), Applications of Evolutionary Computing: Evoworkshops 2003. Springer.
Keedwell, E., Narayanan, A., Savic, D., 2002. Constructing gene regulatory networks using artificial neural networks. In: Proceedings of the 2002 International Joint Conference on Neural Networks.
King, A., Lu, L., July 2002. A backward analysis for constraint logic programs. Theory and Practice of Logic Programming 2 (4), 517–547.
Knuth, D. E., 1981. The Art of Computer Programming: Volume Two, Seminumerical Algorithms, 2nd Edition. Addison-Wesley.
Knuth, D. E., 1997. The Art of Computer Programming: Volume One, Fundamental Algorithms, 3rd Edition. Addison-Wesley.
Knuth, D. E., 1998. The Art of Computer Programming : Volume Three, Sorting and Searching, 2nd Edition. Addison-Wesley.
Kuipers, B., 1994. Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge. MIT Press.
Lamb, T., Wischik, L., 1996. Walk: A stochastic simulation of the G-protein cascade of phototransduction, available on the WWW at `http://www.physiol.cam.ac.uk/staff/lamb/Walk/index.html` (visited September 2000).
Lamb, T. D., 1996. Stochastic simulation of activation in the G-protein cascade. Biophysical Journal 67, 1439–1454.
Lee, M., 1999. On models, modelling and the distinctive nature of model-based reasoning. AI Communications 12, 127–137.
Liebovitch, L., Tao, Y., Todorov, A., Levine, L., 1996. Is there an error-correcting code in the base-pair sequence of DNA? Biophysical Journal 70 (2), MP017.
Lutz, M., Ascher, D., 1999. Learning Python. O'Reilly.
Mendes, P., 1993. Gepasi: A software package for modelling the dynamics, steady states and control of biochemical and other systems. Computer Applications in Biosciences 9, 563–571.
Mendes, P., 1997. Biochemistry by numbers: simulation of biochemical pathways with Gepasi. Trends in Biochemical Sciences 22, 361–363.
Mitchell, M., 1996. An Introduction to Genetic Algorithms. Series in Complex

Adaptive Systems. Bradford Books/MIT Press.

Monks, C. R., Freiberg, B. A., Kupfer, H., Sciaky, N., Kupfer, A., September 1998. Three-dimensional segregation of supramolecular activation clusters in T-cells. Nature 395, 82–86.

Morton-Firth, C., Bray, D., 1998. Predicting temporal fluctuations in an intracellular signalling pathway. Journal of Theoretical Biology 192, 117–128.

Narayanan, A., Keedwell, E. C., Olsson, B., 2002. Artificial intelligence techniques for bioinformatics. Applied Bioinformatics 1 (4), 191–222.

Nemirovsky, A. M., 1994. Is Schrödinger's cat object-oriented, IBM Report.

Nielson, F., Nielson, H. R., Hankin, C., 1999. Principles of Program Analysis. Springer.

Norton, C. D., Decyk, V. K., Szymanski, B. K., 1996. On parallel object oriented programming in Fortran 90. ACM SIGAPP Applied Computing Review 4 (1), 27–31.

Norton, C. D., Szymanski, B. K., Decyk, V. K., 1995. Object-oriented parallel computation for plasma simulation. Communications of the ACM 38 (10), 88–100.

Reddy, V., Liebman, M., Mavrovouniotis, M., 1996. Qualitative analysis of biochemical reaction systems. Computers in Biology and Medicine 26 (1), 9–24.

Reeves, C. R. (Ed.), 1993. Modern Heuristic Techniques for Combinatorial Problems. Blackwells.

Ricard, J., 1999. Biological Complexity and the Dynamics of Life Processes. Elsevier.

Salomon, D., Gullick, W., 2001. The erbB family of receptors and their ligands: multiple targets for therapy. Signal 2 (3), 4–11.

Sauro, H., 1993. SCAMP: a general-purpose simulator and metabolic control analysis program. Computer Applications in Biosciences 9 (4), 441–450.

Sauro, H., Fell, D., 1991. SCAMP: A metabolic simulator and control analysis program. Mathematical and Computational Modelling 15 (12), 15–28.

Schlessinger, J., 2000. Cell signaling by receptor tyrosine kinases. Cell 103, 211–225.

Sigurgeirsson, H., Stuart, A. M., Wan, W.-L., 2001. Collision detection for particles in a flow. Journal of Computational Physics 172, 766–807.

Skiena, S., 1997. The Algorithm Design Manual. Springer.

Smith, G., Sternberg, M., 2002. Prediction of protein-protein interactions by docking methods. Current Opinion in Structural Biology 12, 28–35.

Stevens, P., 2000. Using UML. Addison-Wesley, original edition 1999.

Stroustrup, B., 1991. The C++ Programming Language, 2nd Edition. Addison-Wesley.

Tomita, M., Hashimoto, K., Takahashi, K., Shimizu, T., Matsuzaki, Y., Miyoshi, F., Saito, K., Yugi, K., Venter, J., Hutchinson, C., 1999. E-CELL: software environment for whole cell simulation. Bioinformatics 15 (1), 72–84.

Wagner, A., Fell, D. A., 2001. The small world inside large metabolic networks.

Proceedings of the Royal Society of London B 268 (1478), 1803–1810.

Watts, D. J., 1999. Small Worlds: The Dynamics of Networks between Order and Randomness. Princeton University Press.

Whalley, J. L., Tuite, M. F., Johnson, C. G., 2002. A virtual lab for exploring the $[psi]^+$ yeast prion. In: Valafar, F. (Ed.), Proceedings of the 2002 International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences. CSREA Press.

Winder, R., Roberts, G., 2000. Developing Java Software, 2nd Edition. Wiley.

Yarden, Y., Sliwkowski, M. X., February 2001. Untangling the ErbB signalling network. Nature Reviews: Molecular Cell Biology 2, 127–137.

Analysis	**Design**	**Implementation**

A receptor protein	At a particular time a receptor protein is at a particular position in the cell and has a particular velocity. It is either phosphorylated or not. It is able to move at its current velocity, change velocity, or become (de)phosphorylated.	`class receptorProtein:`
		
		`decimal x,y;`
		`decimal dx,dy;`
		`trueOrFalse phosphorylated;`
		
		`behaviour: createNewProtein`
		`x = 0.0; y = 0.0;`
		`dx = 0.0; dy = 0.0;`
		`phosphorylated = false;`
		
		`behaviour: moveProtein`
		`x = x + dx;`
		`y = y + dy;`
		
		`behaviour: changeVelocity(newDX, newDY)`
		`dx = newDX;`
		`dy = newDY;`
		
		`behaviour: phosphorylate`
		`phosphorylated = true;`
		
		`behaviour: dephosphorylate`
		`phosphorylated = false;`

Table 1
An example of the end result of the three parts of the modelling stage: analysis, design and implementation.

Concept	**Description**
Class	A *class* is the representation in the computer model of a type of thing found in the real system being modelled.
Object	An *object* is a representation of a particular object being simulated. Each object belongs to a *class*, which says what kind of thing it is. There can (usually) be as many objects of a particular class as are needed by the simulation, limited only by memory constraints in the computer.
Attributes	The *attributes* of a class is the set of data needed to describe an object of that class.
Methods	The *methods* of a class are the descriptions of how objects of that class can behave: how they can change, how they can modify other objects, the kinds of information that they can communicate, et cetera.
Analysis	The phase in the construction of a computer program where the real world problem is studied and broken down into a number of interacting classes.
Design	The phase in the construction of a computer program where the classes and interactions developed in the analysis stage are converted into computational data structures and ways of modifying those structures.
Implementation	The phase in the construction of a computer program where the structures and interactions developed during the design phase are translated into a computer language.
State	The *state* of an object is the current values of the attributes in the object.
Inheritance	One class (the *subclass*) is said to inherit from another (the *superclass*) when it takes the methods and attributes from the superclass and modifies them to form a new class. This is important for a number of reasons: e.g. it is possible to rapidly develop new classes by modifying existing ones, and it is possible to substitute any of the more specialized subclasses for the superclass.

Table 2
Summary of core concepts in object-oriented modelling.