Complex Systems Models: Engineering Simulations

Fiona A. C. Polack[1], Tim Hoverd[1], Adam T. Sampson[2], Susan Stepney[1], and Jon Timmis[1],[3]

[1]Department of Computer Science, University of York, UK, YO10 5DD
[2] Computing Laboratory, University of Kent, Canterbury, UK, CT2 7NF
[3]Department of Electronics, University of York, UK, YO10 5DD
Fiona.Polack@cs.york.ac.uk

Abstract

As part of research towards the CoSMoS unified infrastructure for modelling and simulating complex systems, we review uses of definitional and descriptive models in natural science and computing, and existing integrated platforms. From these, we identify requirements for engineering models of complex systems, and consider how some of the requirements could be met, using state-of-the-art model management and a mobile, process-oriented computing paradigm.

Introduction

In computing contexts, and particularly the context of artificial life, complex systems are studied through computer simulation. Reynolds' boids (Reynolds, 1987) is a classic example, where the complex flocking or swarming behaviours are shown by visualisation of a large number of simple boid processes obeying simple rules.

Simulations are used to model complex systems – biological phenomena, economies, human societies, and much more. Typically, a simulation is built to explore a specific problem in a specific context; there is little attempt to develop generic solutions, or to record any design or engineering. Often a valid simulation is judged to be a model that produces the expected results by a process that looks a bit like reality; there is little concern for the quality of the underlying simulation (Epstein, 1999). General support for complex systems and agent modelling tends to be at the implementation level (see, for instance, the ACE resources, www.econ.iastate.edu/tesfatsi/ace.htm). An immediate result of this focus is a long-running intellectual debate about whether it is possible to do science through simulation (see Miller (1995); Paolo et al. (2000); Wheeler et al. (2002); Bryden and Noble (2006)). Similar issues with the validity of simulation evidence arise in safety engineering and other dependability, assurance (Alexander, 2007). For ALife, it has already been noted (e.g. by S. Bullock, in Wheeler et al. (2002)) that to assess the role and value of complex systems simulation, we need to address deep questions of comparability: we need a record of experience, of how good solutions are designed, of how to chose parameters and calibrate agents, and, above all, how to validate a complex system simulation.

The sorts of systems in which we are interested are complex in the sense of having elaborate behaviour at a high level that is the consequence of many simple behaviours at a lower level. The high-level behaviour cannot be deduced as a simple combination of low-level behaviours – in the same way that the velocity of a flock of birds is not derivable by any simple analysis of the behaviours of the individual birds. Space, time and the environmental context are critical features of these systems. Engineering of such complex systems requires support for the software engineering of computer simulations, for use in the investigation of complex systems in nature, *in vitro* and *in silico*.

Ultimately, our goal is to engineer simulations of systems exhibiting several layers of emergence – the lowest level gives rise to emergent behaviours at an intermediate level, and the ensemble of these behaviours gives rise to further behaviours at still higher levels (see Turner et al. (2007); Stepney et al. (2006); Polack et al. (2005)). We see this as an essential feature of initiatives such as molecular nanotechnology that aim to engineer interventions in natural and complicated systems through management of emergent properties; our work is also relevant to macro-scale complex systems – often referred to as systems of systems – such as human organisational systems, traffic management.

This paper reports an initial investigation into the state of the art in complex system modelling and software engineering, that leads to a consideration of how existing approaches and techniques can be adapted and used for engineering simulations of complex systems. We consider some interdisciplinary approaches to modelling and simulating complex systems that adopt software engineering models and tools to describe complex natural systems. We identify advantages of these models, but also their failure to adequately express and manage emergent properties.

The state of the art in software engineering of simulations for systems biology comprises a number of interdisciplinary projects that integrate modelling tools and visualisation facilities, to construct specific, flexible platforms for experimentation. The review shows that these projects have many of the expressive features needed for a platform, but that they do not necessarily generalise in the ways that we wish. As a first step towards a general simulation platform, we consider how advances in software engineering might help.

Models of Complex Emergent Simulation

A *model* is an abstraction that is made to aid understanding or description of something. We can distinguish two orthogonal modelling goals: description and definition. For a complex emergent system, a descriptive model might capture aspects of the observed high-level behaviour; in modelling natural systems, scientists use models to capture what they observe. A definitional model is more typical of conventional engineering – it expresses required characteristics of a system at an appropriate level of abstraction. A definitional model can be refined, translated and analysed, to improve understanding of system characteristics, and, in engineering, to support construction of an artificial system.

Here, we consider some existing approaches, divided into mathematical models and diagrams. Both include models that are descriptive and models that are definitional. We then consider existing tool support for these approaches. Finally, we look at two state-of-the-art approaches that combine existing modelling and tool support. Our aim is to postulate requirements for engineering simulations of complex systems, through identification of good practice in explanatory and exploratory simulation of complex systems.

Mathematical Models

In science, mathematical models are essentially descriptive, attempting to replicate observed aspects of natural structure or behaviour. Physics and biology often use differential equations to approximate the observed behaviour of a high-level system, based on continuous variables at a lower level. For example, the Lotka-Volterra differential equations are important for modelling predator-prey systems. Stochastic models (for instance, Monte Carlo simulations) also aim to capture the high-level behaviour of complex systems.

The scientific use of mathematical models is instructive; the models allow scientists to explore variables that might contribute to observed behaviour. Once candidate variables are selected, the hypothetical result of changing the values or relative importance of variables can be studied. The best mathematical models provide convincing evidence that the modelled variables do indeed influence the real behaviour. These models also provide benchmark results: a simulation that produces realistic observable behaviour should also produce data. Mathematical models could form a basis for evaluating simulation-derived data against real-world data.

However, in the context of complex systems engineering, there are several limitations to the scientific use of mathematical models. The models rely on already having identified the key system components; furthermore, there must be an objective, typically discretised, representation of those components. In the real world, emergent behaviour does not arise through solving differential equations; these models are analogues, but do not provide significant insight into the continuous internal process of a complex system. Furthermore, the scientific models are not definitional – they do not directly admit engineering refinement or analysis.

In software engineering, definitional mathematical models use discrete mathematical concepts, from set theory, predicate logic, etc. Models are formalisations of programming concepts such as the Hoare logics (Hoare, 1969) and Dijkstra's predicate transformers (Dijkstra, 1975). Referred to as formal specifications, the models capture the structure, behaviour and/or communication protocols of systems, and provide the basis for various analyses of correctness.

Diagrams

Historically, biological illustration uses bespoke, informal sketches to express observed relationships or interactions, without any systematic notational definition. More recently, modelling techniques have been adopted from other disciplines – systems biologists, and their interdisciplinary collaborators, are turning to existing diagrammatic notations with defined syntax (and sometimes defined semantics). The use of diagrams is still largely descriptive, even though the notations originate in the definitional context of software engineering. Three classes of diagram can be distinguished.

Connectivity diagrams express the known connectivity of natural systems, using analogies to electrical circuits or software components. Examples include circuit diagrams, interaction diagrams, and various message sequence charts. Connectivity diagrams map well to mathematical languages – process algebras have been used to model many aspects of cells (and other biological systems) and to formally express and analyse communication protocols.

Structural, or class, diagrams describe static components and their relationships. The current fashion is to use class diagrams, where a class is an intensional definition of some local data (variables, constants) and the behaviours needed to maintain that data. The extension of a class is an object, that holds specific instances of data. The associations of a class determine how objects of various classes can interact; associations can be thought of as providing the potential for connectivity, whilst class behaviours include those needed to establish and maintain connectivity among objects.

An important, and biologically attractive, aspect of class diagrams is that the classes and associations represent families of conformant instances (objects and links, respectively). Thus, a class *cell* represents arbitrarily many similar instances of the cell. Scientists sometimes prefer to capture the structure of specific scenarios, using object diagrams – an object is an instance of a class. In this context, snapshot diagrams can also be used, to express the structural effects of the execution of methods or operations on objects.

A problem with structural diagrams for complex system modelling is that there is no sense of the system as an entity – the system view is a collection of type descriptions. We can constrain the number of objects that are linked to each object of another class, but we cannot easily define how many objects exist (relatively or absolutely; the number of objects may be highly dynamic). Furthermore, we can define methods to create and destroy instances, but these can only be constrained by static predicates, not by system-wide observation or dynamic preconditions (how many are needed, or how many can be supported by the current environment).

State machines are essentially variants on (finite) state automata. They express the possible evolutions, either of an object or of a system as a whole. Object-level diagrams have the advantage of simplicity – interaction is indicated by shared events or generation of events to other state diagrams. There are many notations and variants, including Petri nets, Harel state charts, UML state diagrams.

A state machine defines, firstly, the different states of existence of an object (or system). In current realisations of state machines, a state is distinguished by the applicable range of values of its variables (if a state machine relates to objects of an object-oriented class, states are defined over attribute values). Next, the state machine defines the ways in which an object can change state, via transitions. A transition is a response to an event, and an event is, typically, an input received by the object (or system). Transitions are protected by guards – a set of conditions, concerning the wider system state (and perhaps the environmental context of the system) that must be true if the state is to change. Semantically, a state machine may require the state to change whenever an event is received and the guards are true, or, less-commonly, it may simply permit the state change.

An advantage of state machines for biological systems is that they can express known stimuli and responses. Most state machine notations admit concurrent states, which, with the ability to capture incoming and outgoing events, make it possible to construct sophisticated models of, for instance, cell interaction. The diagrams express potentially-dynamic structures, and can provide drivers for simulation of collections of objects. However, the same limitation arises as on class diagrams: the number of objects that are operational at any time cannot be defined in the models.

Tools for Models

In computer systems engineering, and in scientific description, modelling is increasingly tool-driven: use of models generally means use of modelling tools. In computer science, tools support formal specification (definitional mathematical modelling), providing type-checking and proof assistance. Proof can be applied to conjectures about a model and about the relationships between models (refinements, retrenchments, reifications). In natural sciences, descriptive mathematical models are equations that simulate behaviour; tools include statistical techniques to assist in identification of variables (used in deriving equations) and in analysis of results. Tools to solve equations (heuristic or absolute) are also common. In both contexts, tools usually support a single language, and generally require some expertise.

Tools for diagrammatic modelling tend to be commercially-driven. Usability, in high-productivity commercial contexts, takes precedence over strict conformance to standards and accuracy. Like mathematical tools, diagramming tools usually support one notation, which is often a proprietary variant of a public, *de facto* or industry standard, with at best limited documentation of less-standard features. (Note that the widely-used UML is one standardised notation that supports many views of a system (http://www.omg.org/spec/UML/2.1.2/).) Traditional tools for diagrammatic modelling support concrete syntax, and may impose some well-formedness conditions. The ability to check well-formedness has improved significantly in recent tools aligned to management of models; the ability to refine and analyse diagrammatic models is also improving. We return to this aspect of tool support later.

A Brief Review of the State of the Art

Rather than attempting a review of complex systems modelling in general, we consider two state-of-the-art approaches, noting their strengths and limitations.

Perhaps the most advanced computer contribution to the simulation of real biological systems is currently found in Reactive Animation (RA) (Efroni et al., 2007; Sadot et al., 2007), an approach that combines off-the-shelf tools into a sophisticated and flexible simulation environment. The key modelling components are Rhapsody statecharts (state machines) and Live Sequence Charts (connectivity diagrams). The authors describe their work as reverse-engineering biological systems into protocols and object-evolution models. Experimentally-derived (real) biological data is used to populate the initial state of a simulation. Among the facilities for interacting with and manipulating the simulation are adjustable biological-scale time, and zoom-in and tracking facilities. It is also possible to adjust the underlying models and see the effects directly on the simulation.

A key aspect of RA is its modularity: the modelling tools are separate, integrated through the *InterPlay* application, and manipulated through a *PlayEngine*. Similarly, the systems that are modelled can be composed in a modular way. Clever integration means that modification to simulations can either be initiated through the interface and reflected in models, or initiated in models and reflected in the interface.

RA comes from an interdisciplinary team, with leading researchers from several communities bringing their complementary skills and problems. Although the integration is modular and thus flexible, the current work is closely tied to proprietary modelling tools. RA is an existence proof that integrated, flexible simulation and modelling is possible, rather than a general solution to modelling and simulation of complex systems. Also, the motivation for the work is to model a complete organism; our more general motivation is to support the engineering of complex systems. Knowing how to replicate the behaviour of a complex system, and being able to extend our knowledge, are critically important, but are only part of this wider motivation.

The second example of state-of-the-art modelling of complex systems comes from the process algebra community. PEPA (Calder et al., 2006, 2008) uses stochastic process algebra to construct complementary models of a biological network – a reagent view (perhaps akin to the state machine models) and a network view (akin to the connectivity models). The reagent view can express concentrations and triggers to biochemical product formation, whilst the network view captures time-ordered sequences of events across the system. Whilst diagrammatic views are supported, the PEPA modelling is strictly mathematical; the views use the same mathematical language, and have been proved isomorphic. The formalism supports proof of properties – proof of deadlock-freedom, for instance, improves the confidence of the modellers in their networks, since nature does not normally exhibit the forms of deadlock that we observe in communicating (computational) systems.

Like RA, PEPA was developed in a well-integrated interdisciplinary context, to help researchers understand the biological networks that they could observe and measure in the laboratory. The PEPA workbench (http://www.dcs.ed.ac.uk/pepa/tools/), which supports property expression and proof, also supports an algorithmic approach to generating conventional ordinary differential equations from the PEPA models, which allows a clear comparison of observed behaviour of the system represented by the models with results of laboratory analysis.

PEPA (and other process algebra approaches such as bioAmbients (Regev et al., 2004)) demonstrates the benefit of deep integration. The models are different representations in the same notation, with a common semantics. The approaches work well in closely-coupled interdisciplinary contexts, where experts in process algebra work alongside laboratory scientists. However, experts in process algebra are not particularly common, even in Computer Science. Like the proprietary-tool buy-in of RA, PEPA is an existence proof for simulation environments, rather than a general solution to modelling complex systems that would be amenable to use by research groups and system engineers without specific expertise.

Requirements for Complex Emergent Systems Design

Whilst the component models of RA and PEPA are definitional, the goal of these simulation initiatives, like much complex systems modelling, is descriptive, motivated by a need to express observations of real systems, in order to explain or explore natural processes. In seeking models for engineering complex systems simulations, we need definitional models that are amenable to use by interdisciplinary researchers. We need to be able to relate definitional models (functional requirements and their realisations) to descriptive models that identify what the high-level system should achieve (the emergent behaviours that we want). We start by considering what desirable aspects of complex systems are expressible in the reviewed forms of model. We then consider other requirements and how they might be met.

Existing modelling approaches can express (and define) features such as:

- known structures within and among components – using mathematical relations, or structure diagrams;
- protocols for communication among components – using process algebras or diagrammatic models of interaction;
- potential state changes – using state machines and other variants of state automata.

Each form of model presents a limited view, or a single aspect, of the system. Most of the models are static – they either capture the state of a system or they prescribe possible histories of a system. None is really explanatory, in the sense of providing understanding of the layered processes that determine a particular complex system. For engineering complex systems, we need models that,

- express the characteristics of multiple instances of low-level systems, as well as the required emergent characteristics of high level systems;
- represent the context (in terms of space, time and relevant environmental features) of systems;
- capture the cumulative make-up of systems (quantities of objects etc.).

Where models of natural systems are used as a basis for simulation, it is sometimes the case that, rather than model knowledge about a natural system directly, the diagrams express a software engineering design or aspects of the implementation – natural concepts are modelled with computing-related attributes (`name: string`) and operations (e.g. `print()`). Both natural and design models are necessary, but there is a need to be explicit about the purpose of a model, and there is a need to understand and express correspondence between the two sorts of model.

In addition to general engineering needs, we can divide other requirements into: features of complex systems that are not met by existing approaches to modelling; and desirable features of models.

System Features Not Covered

The key omissions, for accurately capturing the range of views of a complex system, can be summarised as dimensionality and scale.

Dimensionality must be considered, since a complex emergent system is, by definition, concerned with time – the emergent properties emerge when the system runs for a period of time. In most cases, a complex emergent system is also concerned with space, since the separation of components fundamentally affects their ability to interact.

Scale can mean two things. Firstly, the relative or absolute quantities of components in a system can affect whether emergence occurs (what the system actually achieves). As noted above, diagrams of the system state do not have an obvious way to define the quantities of objects created, or to define dynamic constraints on behaviour (other than through transition guards).

The second meaning of scale concerns the scale of observation. This is what determines the subject of models: the emergent system or the system components. Conventional engineering models operate at one observation scale, so the diagrams (even in combination) cannot be used to explore inter-level effects such as emergence or self-organisation.

Scale of interaction is critical. Complex behaviour arises when many (hundreds, thousands, or even billions) instances interact. The models typically used in systems biology, and in their conventional electronics and computer science origins, express constraints on interaction, but cannot express the cumulative interaction that is the root of emergence.

Furthermore, the emergent characteristics of a complex system are typically the result of interaction across scales of observation – low-level components induce local effects on their environment; higher-level components monitor their environment and thus react to changes, once the cumulative local effects are detectable at the higher level.

Desirable Features of Models

We can identify a number of desirable features for engineering models of complex systems; these features are often apparent in the modelling forms that already exist.

Modifiability is essential if models are going to be effective tools in engineering or scientific research. It is highly desirable that modifications in one view or instance of a model are reflected (automatically) in other views. In all areas, modification is used to adjust models to meet some external criteria (e.g. realism, customer requirements, etc.) In engineering, modification also means *translation* from an abstract model to an implementation level, or between notations that are in some sense equivalent. The problems of consistency under modification have challenged designers for many years, and are exacerbated by inconsistent or unintegrated modelling tools and ill-defined notations.

Understandability has many aspects, but in complex systems it tends to rely heavily on visualisation – photographs, sketches, diagrams, mathematical formulae, simulations. However, the understandability of visualisations depends on the ability of the reader to interpret the visual forms in the ways intended by the author of the model. We need ways to express and encourage shared interpretations.

There have been many (very many) works defining the meaning of model elements, and semantics is widely studied. For static models, it is useful to distinguish the specific language of the model (the visualisation – the shapes of components, location of labels etc), from the abstract concepts (the underlying model) – hence concrete and abstract syntax. Furthermore, it is useful to provide a definition of the meaning of the concept (semantics), for instance by reference to a well-understood or better-defined concept (so, mathematical sets can represent the semantics of the data aspect of a class of objects). However, an area that is less well studied is behavioural semantics. We do not have well-defined ways to develop models, or well-defined ways to interpret what static models tell us about the temporal and spatial behaviour of the systems that are modelled.

Towards Meeting the Requirements

Reviews of existing modelling approaches and their possible contributions to the engineering of complex system simulations demonstrate some ways in which the requirements can start to be addressed. Indeed, state-of-the-art modelling of natural systems has already provided bespoke solutions in restricted contexts. We first consider ways in which the required system features might be addressed in engineering methods. Then, to address requirements for features of models, we outline ways in which model integration and model management might be used to support the integrated tool platforms needed to engineer complex system simulations.

Requirements for Coverage of System Features: Exploring Simulation Environments

As demonstrated in the RA approach, above, time and space are inherent to simulated models. The RA simulations are derived by a specific form of execution of the static object and state machine models on multiple (diverse) instances simultaneously. The visual simulation shows the emergent effects across time and space. In modelling terms, the static models represent potential point-in-time observations of single objects in a collection. The simulation is then a simultaneous running of many possible paths through the static models.

A simulation environment for engineering complex systems needs to be able to relate simulations and static models in ways that are not commonly attempted. Simulation environments need to be able to constrain the simulation to follow the static models, but to free the simulation from biases – accidental constraints imposed by over-eager modelling or by the simulation environment itself. A common form of over-constraint is the use of absolute spatial co-ordinates –

in natural complex systems, components have only local reference, to their nearest neighbours; there is no component-level view of the whole system. If a simulation locates components by absolute co-ordinates, locality becomes a derived attribute, not an inherent concept. In general, such simulations are inadequate because emergence due to local interaction is masked by unnatural global effects. Furthermore, simulations using absolute spatial co-ordinates pose engineering problems: they are hard to distribute and hard to extend dynamically, because the spatial algorithms are hard-coded.

When simulating complex systems, it is difficult to avoid bias in the execution. This can be illustrated at various levels. Simulated systems of differential equations often display some realistic-looking behaviour. However, the behaviour is significantly biased by the way that the equation is constructed (the formula selected) and the variables and constants chosen. Next, spatial emergence can be shown in simulation, but the form of spatial emergence is typically biased by the form of underlying representation. Consider systems such as "game of life" cellular automata, where the representation is usually shading of cells in a regular, two-dimensional grid: changing the grid or the shading can make a significant difference in the perceived emergent behaviours. Finally, there are biases related to metrics – the number of instances, the time step, the spatial granularity, even the number of time-steps over which a characteristic is measured.

A key question in the use of simulation for engineering complex systems (or for understanding naturally-occurring complex systems) is – how far down must we go to avoid bias? In most cases, we do not have to go as far as a simulation of quantum mechanics for a meaningful biological simulation, but it is a sobering thought that even macro-scale complex systems (such as a galaxy, or at a more tangible scale, traffic flows or distributed command and control systems) are subject to the fundamental laws of physics.

From this discussion, two things arise. The more obvious is the need to integrate static modelling with simulation. Perhaps less obvious is the need for high-performance, flexible simulation environments, which support local reference (to avoid building in biases such as unrealistic long-range communication structures), and can handle appropriate numbers of interacting instances.

It is arguable that an interaction model, to statically express the ways in which instances interact, is missing from the existing approaches. Formal approaches (based on communicating process calculi) and associated informal visualisations (based on models of the connectivity diagram type) can be extended to express this aspect. We need to understand how to integrate on this scale.

Requirements for Features of Models: Managing Models

Recent advances in the theory of and tool support for model management are predicated on the use of modelling tools. The issues that arise in management of mathematical and diagrammatic models relate both to the conceptual basis of the models, and to the ways in which models are used.

In computing contexts, model integration and model comparison are becoming important – often driven by commercial imperatives when organisations (tool developers or client users) merge, but also motivated by academic interest in patterns and commonalities, and by the simple availability of the computational resources to manage collections of models. The solutions being explored are not one-off integrations (in the sense exemplified by RA and PEPA), but generic bases for integration and model management.

Where groups of models are used to express features of a system, the interaction of models is often overlooked, even by professional designers of modelling languages. The different views of a system overlap, and the overlap needs to be consistent. A classical case is where a class diagram and state machine are used together – the states in the state machine should be expressed in terms of attributes shown in the class diagram, and the transitions should be effected by invoking operations of the objects of a class, in accordance with the association structures defined in the class diagram. Similarly, where connectivity diagrams are used, it is important that the internal methods and links are consistent with those in the class diagram, and the sequences of communication are consistent with those permitted in the corresponding state machines. Another classical angle from Computer Science is that, where formal models exist alongside diagrammatic models, there is an obligation to demonstrate the continuing equivalence of concepts – there is a need to define correspondences, or traceability links, between different model concepts.

Unless we understand, and can characterise, the semantics of the engineering models that we adopt, we cannot adequately address adaptation of models to the needs of complex systems engineering – in simple terms, we need to understand the classical concepts of state, transition and class, in order to find sensible ways of accommodating space, time and environment.

Recently, software engineering has considerably advanced the management of models; two movements are establishing fundamentally-comparable definitions of modelling languages. The *unified theory of programming* movement (see Hoare and He (1998); Woodcock (2003); Cavalcanti and Woodcock (2006)) is seeking common mathematical underpinnings for modelling and programming languages. Separately, commercially-driven research into model-driven development (Swithinbank et al., 2005) led by organisations such as the Object Management Group (http://www.omg.org/), and large international projects such as the EU Modelplex initiative (https://www.modelplex.org/), focus on defining common abstract concepts for modelling languages, and domain-specific capabilities, through the use of *metamodels*. The associated model management concepts have the ability to support formal as well as diagrammatic languages.

Both initiatives open new ways to compare, validate, extend, and transform models. Here, we focus on model-driven development (MDD), because it is more accessible to developers.

Model-driven development seeks to manage the consequences of distributed development: developers in large projects typically produce different variants of many types of models that must be shown to express equivalent concepts; furthermore, having produced integrated design models, implementation can be made faster and less error-prone if repetitive programming is automated from design models.

The fundamental concept of MDD is the model stack: a particular realisation (M_0 level) is an instance of a model (M_1 level), whilst that model is an instance, or realisation, of a metamodel (M_2 level). The current top of the stack defines languages for metamodelling (M_3 level). The four layers are not always clear-cut, but they provide a reasonable separation of concerns and a sufficient basis for model management. MDD tool suites such as Epsilon (http://www.eclipse.org/gmt/epsilon/, Kolovos et al. (2006)) support expression of models and metamodels, checking of models against metamodels (for syntactically-correct use of notations), and checking of consistency across models (that all models in a family use concepts in the same way, at both the notational and application levels). These tools support model transformation, with mappings between concepts in different metamodels being applied to transform models, either between notations, or from design models to implementations.

Predominantly, MDD has been used for diagrammatic models, annotated with quasi-formal constraints. However, recent work adds consideration of text models (programs, formal texts, meta-language texts). This raises the intriguing possibility of creating general integrations, to exploit whatever models or tools are suitable for a particular system, or to a particular research group's expertise.

Discussion

In common with Cohen and Harel (2007), we take the view that complex emergent systems cannot be constructed (or simulated) solely as hierarchies of sequential transformations. We must capture the concurrent, reactive nature of these systems. More explicitly, however, we must recognise the importance of scale dependencies and interactions across scales – the many concurrent inputs that Cohen and Harel (2007) observe are themselves the results of many concurrent outputs at other scales. In the reviewed approaches (RA, PEPA), the teams have, by chance or with great skill, selected areas of study where one or two adjacent levels, or scales, can be modelled to give realistic results that match the level of observation of the research scientists concerned.

In researching a broad platform for complex systems simulation, we would exploit work in mobility and process algebras. The inclusion of mobility allows the modelling of processes that dynamically change their relative location by changing their channels of communication with other processes, affording an effective way to model the structural plasticity within systems. One example is the Graphical Stochastic π-calculus (GSπ), developed by Phillips and Cardelli (2007) and proven equivalent to Milner (1999)'s π-calculus. This provides an accessible front-end environment whilst retaining the power of the underlying π-calculus. On a larger scale, the CoSMoS project (http://www.cosmos-research.org/) is building capacity in generic modelling tools and simulation techniques for complex systems; it is predicated on a well-founded process-oriented modelling platform, using the occam-π language. occam-π (http://occam-pi.org/) is a small language that implements the communication strengths of CSP (Hoare, 1985) and the mobile aspects of π-calculus; the well-grounded semantics of these specification calculi provide a formal basis for the programming environment, and support an engineering approach to the underlying mathematics (Barnes and Welch, 2004; Welch and Barnes, 2005). As in PEPA, process algebra can be used to prove properties (such as deadlock-freedom), but, here, the proven properties are then built in to the language (in the GSπ calculus; in the occam-π Kroc compiler) and used implicitly in models produced in the more-accessible front ends (the GSπ environment; occam-π code). Expertise in process algebra is not needed to use these languages, and occam-π can efficiently support millions of concurrent processes, distributed over multiple processors.

A general environment for simulation and modelling of complex systems is about more than concurrent mobile programming. The CoSMoS platform should eventually support many levels and scales – extending upwards (to observe more global effects) and downwards (in search of key causalities and origins). Seth Lloyd (2005) eloquently presented the ultimate simulation: the quantum computer that efficiently simulates the Universe (it is big!). A strict approach to modelling complex systems might expect to start at the very bottom – after all, classical physics emerges from quantum mechanics, and chemistry from classical physics. However, a rational view is that, at each level of interest, the effects of lower levels are of varying importance, and can sometimes be aggregated or omitted without a significant effect on the emergent behaviour. We hope that we will not need to engineer a Universe computer, but to successfully research and engineer complex systems we need tools that helps us to determine the relative importance of lower levels and of views of lower levels.

Finally, we are completely in agreement with Cohen and

Harel (2007) when they state, ... *a computer methodology [sic] that would allow us to zoom back and forth between lower-scale data and higher-scale behaviour while experimenting* in silico *is an ideal way — possibly the only way — to study emergence computationally.* (Cohen and Harel, 2007).

Acknowledgements

This work is part of the CoSMoS project, funded by EPSRC grant EP/E053505/1, and a Microsoft Research Europe PhD studentship.

References

Alexander, R. (2007). *Using Simulation for Systems of Systems Hazard Analysis*. PhD thesis, Department of Computer Science, University of York.

Barnes, F. and Welch, P. (2004). Communicating Mobile Processes. In East, I., Martin, J., Welch, P., Duce, D., and Green, M., editors, *Communicating Process Architectures 2004*, volume 62 of *Concurrent Systems Engineering Series*, pages 201–218. IOS Press.

Bryden, J. and Noble, J. (2006). Computational modelling, explicit mathematical treatments, and scientific explanation. In *Aretificial Life X*, pages 520–526. MIT Press.

Calder, M., Gilmore, S., and Hillston, J. (2006). Modelling the influence of RKIP on the ERK signalling pathway using the stochastic process algebra PEPA. *Transactions on Computational Systems Biology VII*, 4230:1–23.

Calder, M., Gilmore, S., Hillston, J., and Vyshemirsky, V. (2008). Formal methods for biochemical signalling pathways. In *Formal Methods: State of the Art and New Directions*. Springer.

Cavalcanti, A. and Woodcock, J. (2006). A tutorial introduction to CSP in Unifying Theories of Programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220–268. Springer.

Cohen, I. R. and Harel, D. (2007). Explaining a complex living system: dynamics, multi-scaling and emergence. *Journal of the Royal Society Interface*, 4:175–182.

Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457.

Efroni, S., Harel, D., and Cohen, I. R. (2007). Emergent dynamics of thymocyte development and lineage determination. *PLoS Computational Biology*, 3(1):0127–0135.

Epstein, J. M. (1999). Agent-based computational models and generative social science. *Complexity*, 4(5):41–60.

Hoare, C. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Hoare, C. and He, J. (1998). *Unifying Theories of Programming*. Prentice Hall.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

Kolovos, D., Paige, R., and Polack, F. (2006). The Eepsilon Oabject Llanguage (EOF). In *Second European Conference on Model-Driven Architecture*, volume 4066 of *LNCS*. Springer.

Lloyd, S. (2005). *Programming the Universe: A Quantum Computer Scientist Takes On the Cosmos*. Vintage.

Miller, G. F. (1995). Artificial life as theoretical biology: How to do real science with computer simulation. Technical Report Cognitive Science Research Paper 378, School of Cognitive and Computing Sciences, University of Sussex.

Milner, R. (1999). *The Pi Calculus*. Cambridge University Press.

Paolo, E. D., Noble, J., and Bullock, S. (2000). Simulation models as opaque thought experiments. In *Articial Life VII*, pages 497–506. MIT Press.

Phillips, A. and Cardelli, L. (2007). Efficient, correct simulation of biological processes in the stochastic π-calculus. In *Computational Methods in Systems Biology*, volume 4695 of *LNBI*, pages 184–199. Springer.

Polack, F., Stepney, S., Turner, H., Welch, P., and Barnes, F. (2005). An architecture for modelling emergence in CA-like systems. In *ECAL*, volume 3630 of *LNAI*, pages 433–442. Springer.

Regev, A., Panina, E. M., Silverman, W., Cardelli, L., and Shapiro, E. (2004). BioAmbients: an abstraction for biological compartments. *Theoretical Computer Science*, 325(1):141–167.

Reynolds, C. W. (1987). Flocks, herds, and schools: A distributed behavioral model (SIGGRAPH '87). *Computer Graphics*, 21(4):25–34.

Sadot, A., Fisher, J., Barak, D., Admanit, Y., Stern, M. J., Hubbard, E. J. A., and Harel, D. (2007). Towards verified biological models. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*.

Stepney, S., Polack, F., and Turner, H. (2006). Engineering emergence. In *ICECCS'06*, pages 89–97. IEEE Computer Society.

Swithinbank, P., Chessell, M., Gardner, T., Griffin, C., Man, J., Wylie, H., and Yusuf, L. (2005). *Patterns: Model-Driven Development Using IBM Rational Software Architect*. IBM Redbook. http://www.redbooks.ibm.com/abstracts/sg247105.html?Open.

Turner, H., Stepney, S., and Polack, F. (2007). Rule migration: Exploring a design framework for emergence. *Int. J. Unconventional Computing*, 3(1):49–66.

Welch, P. and Barnes, F. (2005). Communicating mobile processes: introducing occam-pi. In Abdallah, A., Jones, C., and Sanders, J., editors, *25 Years of CSP*, volume 3525 of *LNCS*, pages 175–210. Springer.

Wheeler, M., Bullock, S., Paolo, E. D., Noble, J., Bedau, M., Husbands, P., Kirby, S., and Seth, A. (2002). The view from elsewhere: Perspectives on alife modelling. *Artificial Life*, 8(1):87–100.

Woodcock, J. (2003). Unifying theories of parallel programming. In *Logic and Algebra for Engineering Software*. IOS Press.