Choosing and adapting design notations in the principled development of complex systems simulations for research

Fiona A. C. Polack
YCCSA/Department of Computer Science
University of York, York, UK
fiona.polack@york.ac.uk

ABSTRACT

The use of complex system simulation as a research tool is facilitated by principled development; software quality assurance, an important part of fitness for purpose, can be assisted by use of model driven engineering (MDE) techniques. The paper addresses two key aspects of MDE for simulation development: the choice of appropriate modelling languages, and language adaptation, illustrated from a cell division and differentiation simulation development, for use in research on prostate conditions. The resulting language has potential for general modelling of reactive and transitional systems.

Keywords

simulation, complex systems, modelling, metamodel adaptation, multi-level modelling

1. INTRODUCTION

Computer simulation is a potentially useful tool in research on so-called complex systems. Modelling and simulation are used to explore ideas, generate experimental hypotheses, test hypotheses not amenable to experimentation, and predict possible futures. The power of computer simulation as a research tool depends on engineering the system in a way that engenders confidence in users. Part of the demonstration of fitness for purpose of a simulator is evidence of software quality: users must understand the mapping from the domain to the simulator, and any compromises made in design. The use of model-driven engineering (MDE) techniques promotes quality. This paper focuses on the choice and adaptation of modelling languages, drawing on the design of an agent-based simulation of cell division and differentiation. To motivate MDE use in simulation development, the background to principled simulation development is summarised in Section 2. The need to address fitness for purpose has led to creation of a principled process for modelling and simulation, and this in turn focuses attention on the reality gap that is addressed by domain specific modelling languages (DSLs). Section 3 considers selection of modelling approaches for simulation development. Sections 4 and 5 explore and illustrate the requisites for seamless development using MDE. The paper concludes (Section 6 and 7) with a short discussion and conclusions.

2. CONTEXT AND MOTIVATION

Understanding or control of complex behaviour of living systems is hampered by the difficulty of observing the internal behaviour of living organisms. Traditionally, dead systems are dissected and analysed, with no direct observation of internal behaviour and dynamics. Recent non-invasive imaging techniques allow limited observation, but are costly, may damage the organism, and can be difficult to analyse and interpret. Computer simulation provides an attractive, relatively cheap and ethically sound, complement to laboratory techniques. Mathematical models can mimic the dynamics of living organisms or communities of organisms, but to explore the interactions and behaviours that underlie observable dynamic behaviour, agent-based or individual-based modelling is often proposed. The principle of such simulations is that observable behaviours of systems are the emergent (higher level) effects of (lower level) interactions among agents and between agents and their environment.

Whatever the form of modelling and simulation, there is a gap between the simulation and the real system that it attempts to model. This reality gap exists in all engineering: in software engineering, it manifests as the gap between a client's intended goals for a project and the software engineer's system requirements or specification. To narrow the reality gap, researchers have devised sophisticated requirements extraction, analysis, and rapid prototyping techniques. However, in developing a simulation of a complex system for use as a research tool, the reality gap problem extends beyond understanding what the client really wants and explaining what the software engineer can actually deliver. The observable behaviour of many real systems is not fully understood. A complete understanding would encompass many branches of biology and biochemistry, physics and quantum mechanics. In general, it is impossible to completely identify and model all levels and components that determine observed behaviour. The problem is compounded by the many, often conflicting, views of the state of understanding presented in scientific literature: from different laboratories, experiments on different strains or species, different research techniques, and perhaps different scientific conventions. It is generally impractical for a software engi-

neer to attempt to independently determine a coherent and consistent view of the real system.

A model is a limited abstraction or view of the reality. What a software engineer means by modelling may not be the same as what a scientist means. Scientific understanding is often summarised visually using cartoons – that is, diagrams for which the notations do not have, or do not enforce, a well-defined abstract syntax or semantics. Typically, a cartoon tries to express most of the apparently-relevant information about static and dynamic aspects of the system, often over diverse levels of abstraction. Cartoons do not provide an appropriate basis from which to start a software engineering development.

A recent trend is for biologists to use engineering notations: for example, Kitano's bio-regulatory network modelling [12], and circuit diagrams such as Kohn diagrams for signaling pathways [13]. UML variants such as SBML[1], and supporting tool suites, have been tailored to biological use. There are also architecture visualisations such as Matlab's SimBiology [11]. The notations of these diagrams are well-defined, but oriented to real world concepts, not software engineering abstractions. It is often unclear how simulations are generated from the biological views.

2.1 Fitness for purpose

It is rarely possible to know if a model is a sufficient or reliable abstraction. Furthermore, since everything in the model implicitly surrogates for concepts and levels of abstraction that are not included in the model, the way in which complicated real behaviours map to and from the model is not well defined. The reality gap makes it impossible to provide a precise validity assessment of a simulation, but we can capture the basis of confidence in the fitness for purpose of a simulation. Confidence in a computer simulation comes from producing the right sort of results from a system that has the right sort of characteristics, **not** simply from similar measurables. Fitness for purpose must consider the characteristics of the simulator, because complex systems with very different internal behaviour may produce similar observable (emergent) behaviour, and statistically similar results, over at least part of their operational context. A simulation that is deemed fit for purpose in one context may not be a suitable tool for use in a different context, or even for different purposes on the same context.

An argument of fitness-for-purpose [5, 17] expresses the basis of confidence in (a) the science underpinning the simulation, (b) the software engineering development, and (c) the simulation results. Thus, a key part of any argument of fitness for purpose is evidence of software engineering quality, such as appropriate design practices and software testing. Quality cannot rely solely on testing: a simulator cannot completely replicate the effects of the subtle interactions of low-level components, with each other and with the environment, but instead uses probability or stochastic parameters to surrogate very-low-level variability. This means that the simulation produces subtly different behaviour and results on every execution run: writing objective software tests becomes "interesting". Quality assurance instead relies heavily on systematic and thorough development. Many aspects of an agent-based simulation can be developed using conventional software engineering techniques. Agents can be designed and implemented as simple state-and-operations systems, along with their interfaces and interaction. Abstract designs can be transformed into concrete designs and implementations that are valid, in software engineering terms, with respect to the abstract designs. Code can be shown to be correct and consistent at the unit and integration levels, even though its system-level correctness cannot be tested objectively.

The software engineering quality requirements can be met using MDE techniques and principles, within the context of principled development guidelines for simulation.

2.2 Principled simulation development

The CoSMoS process provides principled guidance for development of a simulator that can be argued to be fit for a particular scientific research purpose. The development depends on a trusting collaboration between scientists and software engineers [16]. The collaborators play the roles (in the software engineering sense [15]) of *domain expert* (domain scientists, the end users), *modeller* and *implementer* (software engineers). The CoSMoS process [1, 16] comprises three phases which can be iterated: *discovery*, *development* and *exploration*. Each phase has associated models and a specific relationship to the project goals and roles [1]. The role of MDE relates to the discovery and development phases, and this is the focus of this section.

Throughout discovery and development, the collaborators accumulate information about the *research context* (Figure 1), a repository of shared knowledge about the domain and the development (abstractions, design decisions, motivations, assumptions) that is used in establishing fitness-for-purpose and interpretation of results.

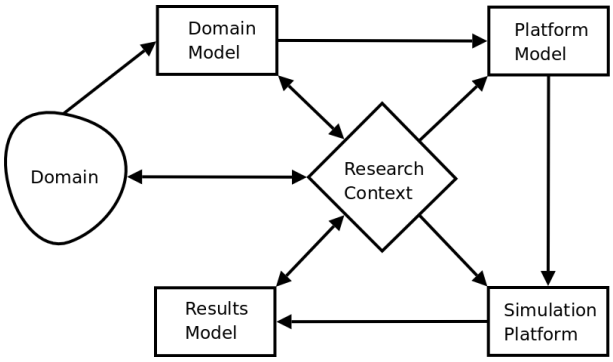

Figure 1: Overview of models in the CoSMoS process, from [1]. Arrows represent flow of information, and should also be associated with validation between models. As in most software-engineering lifecycles, the CoSMoS process can be iterative.

The *discovery* phase focuses on the *Domain* (Figure 1): the collaborators develop mutual understanding of the domain, the purpose of simulation, available information and data etc – papers, cartoons, domain expert commentary, and so on. The domain expert is the domain authority; the software

[1] http://sbml.org

engineers are guided by the domain expert on all questions relating to interpretation of the domain.

The discovery phase also sets up the software engineering development through creation of an agreed *Domain Model* (Figure 1). The domain model is the responsibility of the modeller, who takes information from the domain expert and interprets it into a consistent, well-defined modelling format that is still amenable to checking and review by the domain expert. The ideal is an internally-consistent abstract model of the domain concepts, labeled in the language of the domain, but which supports subsequent systematic software engineering development. Domain modelling is akin to requirements analysis and specification in traditional software engineering, and to development of a platform independent model in MDE. The domain expert plays a role similar to the ideal client in an agile development: in the loop to check that requirements are interpreted adequately, without interfering in software engineering decisions.

In the *development* phase, the modellers and implementers create a *Platform Model* (or platform specific model – PSM) and then a *Simulation Platform* from the domain model (Figure 1). Since the initial domain model is well-defined, the development can use a MDE (or a formal) transformational approach, which has significant benefits in terms of software validation and traceability. To accommodate MDE, we need appropriate use of well-defined modelling languages.

3. MODELLING THE PHYSICAL SYSTEM

Domain modelling requires modelling the physical system. The modellers take domain information and determine which concepts are relevant to the simulation exercise, and how they can be represented. To follow a MDE approach, the models must use notations defined by metamodels. However, to provide the traceability needed to understand the simulation results, it is important to record (in the research context, Figure 1) how the models represent each domain concept and behaviour, and what compromises have been made. The choice of notation for the domain model can facilitate representation of domain concepts, and help to clarify the mapping from real domain concepts to simulation concepts.

In selecting a modelling approach, it is common to "just use UML" [4, 2]. The use of UML may be motivated by use of an object-oriented (OO) language such as Java for implementation: this is a sound software engineering motivation. There are useful extensions to Java for agent-based modelling, such as MASON[2], and, more importantly, it provides thorough design and implementation engineering support through IDEs such as Eclipse[3]. Furthermore the principle of seamless development (using the OO paradigm throughout design and implementation) aids software validation. However, it is not always the case that UML designs are implemented in OO languages: UML uses object and message-passing concepts which do not easily express reactive or transitional systems. It is also the case that not all OO developments take advantage of the quality-enhancing features of IDEs. Furthermore, it is sometimes the case that UML domain modelling ignores the abstract syntax and semantics of UML notations. Sometimes, UML is simply not appropriate for the domain modelling, for instance, Read et al. [19, 18] discuss expressiveness problems of UML class and activity diagrams for reactive systems with feedback loops. Ideally, domain modelling looks for seamless development *from the domain*, as well as seamless development to an implementation.

[2]http://cs.gmu.edu/ eclab/projects/mason/
[3]www.eclipse.org

3.1 Reactive and transitional domain models

In [17, 3], we illustrate domain modelling of reactive and transitional aspects of cell division and differentiation (CDD) using Petri nets and state diagrams. The discovery phase identified that the abstract processes of cell division and differentiation are the key feature of the required simulation, and that these are essentially a reactive system. The domain expert (cancer scientists) proposed modelling with Petri nets (see Figures 2), as these have been much adapted for biological reactive systems (see [6]). Petri nets have been standardised and aligned with model-driven engineering (see, for instance, [10, 9, 7, 8]), giving the potential to develop applications using MDE model operations. However, the CDD domain does not map cleanly to standard Petri nets.

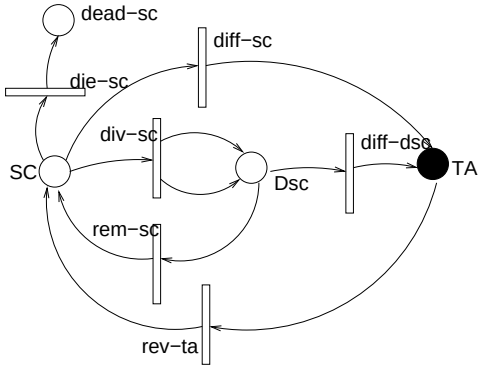

Figure 2: Part of the domain model of the division and differentiation of cells, from [3]. The type of a cell (a token) is determined by the Petri net `Place`: `SC` (stem cell), `TA` (transit amplifying cell – a basal cell), `Dsc` (daughter SC, the result of division), or `Dead-SC`. `TA` is a `RefPlace`, as it also appears on another Petri net. Tokens enter or leave `Places` via `Arcs`. `Transition` names record the source and the form of transition: `diff` for differentiation, `div` for division, `rem` for remain, and `rev` for revert.

In a Petri net, a transition is instantaneous whenever the guarding condition is true, but in the CDD domain a cell may undergo a range of relevant biological processes that affect the state of the cell whilst in one "place". The cell's internal state determines whether, and with what probability, the cell differentiates or divides. The internal abstract states and transitions of cells are a transitional system, and can be expressed in a conventional UML state diagram (or a Harel state chart, though the semantics is not identical).

The purpose of our simulator is to explore the role of cell division and differentiation factors in the neogenesis and control of prostate conditions. The model of a cell must have a persistent representation of the abstract cell state, that

can be used to determine the individual probability of division and differentiation. Relevant changes to the abstract cell state are captured in state diagrams. The cell state is mutable and heritable, to allow exploration of hypotheses of neogenesis. Whilst Petri nets do not have a persistent state, tokens can carry data, and this is sufficient to convey the cell state in the higher level model. Figure 3 shows the state diagram for `SC` (stem) cells.

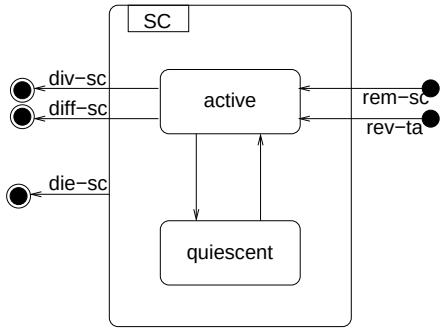

Figure 3: The `SC` state diagram, from [3], uses the transition names from the Petri net, Figure 2. A `SC` arises in the `active` state, via the transitions `rem-sc` or `rev-ta`. The cell may alternate between `active` and `quiescent`, and it can die (`die-sc`) from either state. Exit transitions can only occur from the `active` state.

The CDD domain model has been discussed with the domain experts, who are happy that it is a suitable abstract representation of their laboratory-based understanding – and a suitable basis for generating and testing hypotheses of relating to the development and control of various prostate conditions. The next section explores the variations in the modelling notations required to support MDE development of a simulator from this two-level domain model.

4. PETRI NET SEMANTICS

Standard abstract syntaxes, defined by metamodels, are available for both Petri nets and state diagrams. Modifications are needed to: (a) capture departures from the standard abstract syntax; (b) express linkage of notations; (c) prepare for development of a simulation platform which is an agent model. The modifications are illustrated for the Petri net metamodel only, owing to lack of space. In the Petri net standard [10], metamodels are built up by *metamodel merging* from a core metamodel, part of which is shown in Figure 4.

In adapting the Petri net semantics for the CDD model, we use as a basis the standard *High level Petri net graphs* (HLPNG) extension of the Petri Net core metamodel [8]. The dynamic behaviour of a Petri net is that *tokens* move along `Arcs`, according to `Conditions` expressed on `Transitions`. HLPNGs have `Annotations` recording conditions, and the initial number of tokens in a place (a `Marking`) [8]. The type of a tokens is the `Type` of the place. There are additional constraints, for instance to specify that the `Node` that is the source of an `Arc` must be of a different subclass to that of the target of the `Arc` (a `Place` links to a `Transition`, and a `Transition` to a `Place`). Figure 5 shows the metamodel extract for these annotations.

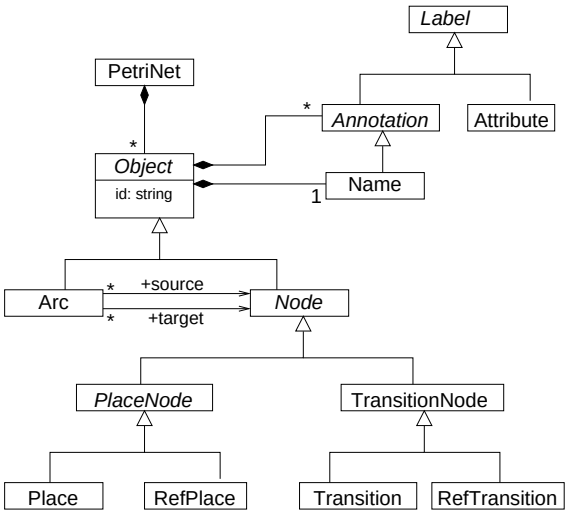

Figure 4: Part of the Petri net core metamodel, based on [10, 8]. A Petri net comprises `Objects`, which may be `Nodes` or `Arcs`. The nodes are `Places` and `Transitions`. Reference nodes and transitions allow the Petri net to be split into sub-diagrams.

5. MODIFYING PETRI NET SEMANTICS

To create a domain specific language (DSL) that helps seamless mapping from domain to domain model, as well as MDE development for the CDD simulator, we wish to extract from and adapt the existing Petri net metamodel and the existing UML state diagram metamodel. We also wish to link the concept of a transition trigger in the Petri nets to the exit transition concept from the state diagram.

A metamodel can be adapted by, for instance, heavyweight extension, weaving in a separate metamodel of the extension, or lightweight decoration of the base metamodel (see [14]). For the CDD DSL, most of the changes to the Petri net metamodel are similar in form to the metamodel merges used in the Petri net standard [10, 8], so a similar approach is illustrated here. Meta-concepts not used in capturing the CDD domain model are omitted (specifically concepts relating to document and graphical layout).

5.1 Transition triggering

In the CDD domain model, a cell can only transition when it is in the right cell state. Thus, Petri net transitions are triggered by a signal derived from the exit transition of a low-level state diagrams.

In the HLPNG metamodel, the `Condition` on a `Transition` is made up of `Terms`, which use: built-in primitives; types, operators and variables defined in the core metamodel; or arbitrary operators and variables defined by the user. An OCL invariant states that the `Condition` must be a Boolean expression [8]. To support the CDD domain model, the Petri net `Condition` is extended to be the conjunction of a standard Petri net `Term` and a (user-defined) `signal` from the state diagram. The signaling semantics could be either to determine which transition fires (e.g. by signaling a specific transition name), or to prompt a probabilistic evaluation of which transition to fire. The propagation of the signal does

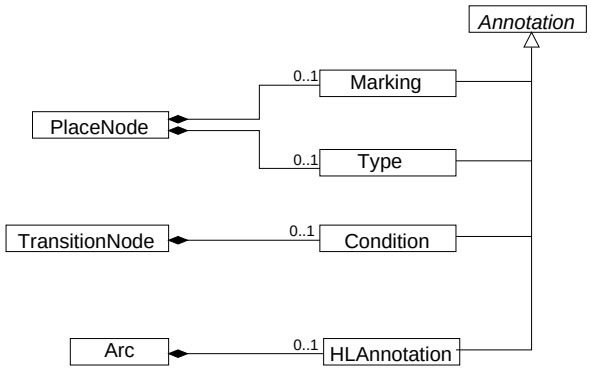

Figure 5: Part of the HLPNG annotation metamodel, based on [8]. This metamodel elaborates the association between `Node` and `Annotation` in the core metamodel Figure 4. When merged with the core metamodel, the annotations on `PlaceNode` and `TransitionNode` also apply to reference nodes.

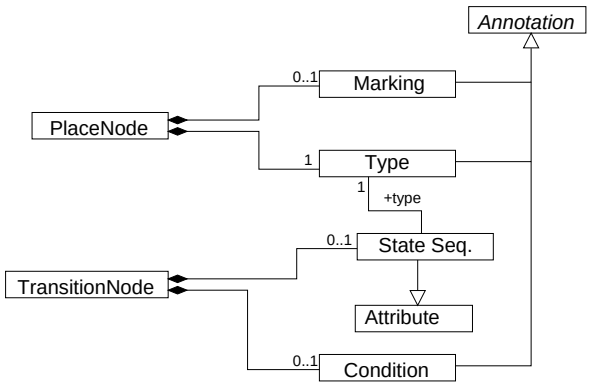

Figure 6: Extending the metamodel of a `TransitionNode`, from Figure 5 to express the associated data store as an `Attribute` (see [8]). `State Seq.` takes its type from that of the associated `PlaceNode` (see text). Note that the `Condition` here is now a conjunction of the HLPNG Boolean expression and an appropriate interpretation of the signal from the state diagram.

not need any elaboration of the state diagram metamodel, as the semantics is the same as that of the existing exit transition.

5.2 Persistent cell state

The Petri net transition semantics (a token is consumed and new tokens produced) need modifying to accommodate mutability and heritability. To model heritability, and to track simulated cells, there needs to be a concept of state and identity that can be passed between tokens across a Petri net transition.

Standard Petri net tokens can to have `attributes`. We add such an attribute that will be instantiated on each token as a *cell-state sequence* that can hold a representation of the abstract cell features that influence cell division and differentiation. The form of the cell-state sequence is determined by the `Type` of the token's current (source) `PlaceNode`. Mutation can affect both the form and value of the cell-state sequence. To pass the cell-state sequence across a Petri net transition, a temporary copy of the cell-state sequence is made, and may be mutated (mutability). The token produced by the Petri net transition has its cell-state sequence initialised from the temporary copy (heritability), using mapping rules as necessary to convert the consumed cell-state sequence type into the produced type.

Figure 6 shows the extension of the HLPNG metamodel `TransitionNode`, adding an association to a new `attribute` subclass, `State Seq.`, to act as the temporary data store. The `State Seq.` takes its type from the source `PlaceNode`. We do not need a second `attribute` to store for the mutated `State Seq.`, because the result of any changes is saved on the produced token, and takes the type of the target `PlaceNode`.

6. DISCUSSION AND FUTURE WORK

This paper focuses on the Petri net metamodel, and notes some minor changes on the state diagram metamodel. The existing UML state diagram semantics allow reference to and modification of values in the cell-state sequence, but do not allow changes to the `type` of the cell-state sequence: whilst in theory type-changing operations could occur at both levels, we allow only Petri net transitions (not state diagram transitions) to change the type of the state sequence. State diagram actions can model the relevant low-level cell behaviour (and time delays), and give a means to capture relevant low-level environmental events: this allows modelling of, for instance, rare cell mutations due to atmospheric radio-activity etc.

The behaviour of the cells that are not in transition on the Petri net needs to be expressed in the linkage of Petri net and state diagram concepts. We do not explore creation of well-defined linkage of the two notations here – metamodels can easily accommodate more than one sort of diagram (consider the UML metamodel), but here, the state diagram model is at a lower level of abstraction, and the link is only the passing of a signal. It is easy to explain the behaviour, but the abstract syntax can only accommodate the connectivity, not define it.

The metamodels do not completely define the CDD domain model: there are still a number of semantic lacunae. Some of the incompleteness is noted above (e.g. the form of the signal from a state diagram). There are also implementational decisions to take, concerning the representation of cells as agents. We have not specified the semantics of agents: such issues require design decisions by modellers or implementers in mapping to a PSM. Because fitness for purpose must be demonstrated, it is necessary to record (as part of the research context) the implications of each design decision, and to understand whether and how it impacts on the fitness of the simulator and the interpretation of simulation results.

To date, all modelling and transformation has been informal and manual, which potentially introduces inconsistencies to the domain model. We would like to use MDE tooling for DSLs to support and validate metamodel modifications. Current DSL development assumes *ab initio* meta-

model creation, but we would like to be able to develop DSLs by extracting from, combining and adapting existing metamodels, as illustrated here. A tool-supported DSL approach would allow us to automate PSM design using MDE model-to-model transformations: we currently have outline PSMs targeting OO implementation and concurrent process-oriented languages.

7. CONCLUSION

The motivation for MDE in simulation for scientific research is to give confidence in the quality of software engineering. We have identified various issues, and shown how some can be addressed at the metamodel level. The specific DSL outlined here gives a well-defined link between state diagrams of component behaviour and Petri nets of system-level behaviour; this has potential use in other reactive and transitional systems.

In future, we would like to be able to create fully-tool-supported DSLs that make MDE development of simulator software easy. Assuming that a well-defined MDE process would address issues of software quality, this would allow attention to focus on addressing the domain to domain modelling gap (for instance by a more systematic approach to notation selection and modification), and on the interpretation of simulation results.

8. ACKNOWLEDGMENTS

The prostate models are from work supported under EPSRC grants EP/F032749 and grant EP/E053505, and by the Wellcome Trust via York Centre for Chronic Diseases and Disorders. The simulator is to complement laboratory work under Maitland's Yorkshire Cancer Research platform grant. I acknowledge my collaborators: Alastair Droop, Philip Garnett, Susan Stepney, Andrei Simionescu, Norman Maitland and the Cancer Research Unit at York. Thanks also to Richard Paige for suggestions on metamodelling.

9. REFERENCES

[1] P. S. Andrews, F. A. C. Polack, A. T. Sampson, S. Stepney, and J. Timmis. The CoSMoS Process, version 0.1. Technical report, Dept of Computer Science, Univ. of York, 2010. www.cs.york.ac.uk/ftpdir/reports/2010/YCS/453/YCS-2010-453.pdf.

[2] H. Bersini. UML for ABM. *Journal of Artificial Societies and Social Simulation*, 15(1):9, 2012.

[3] A. Droop, P. Garnett, F. A. C. Polack, and S. Stepney. Multiple model simulation: modelling cell division and differentiation in the prostate. In *CoSMoS Workshop*, pages 79 – 112. Luniver Press, 2011.

[4] A. J. Flügge, J. Timmis, P. Andrews, J. Moore, and P. Kaye. Modelling and simulation of granuloma formation in visceral Leishmaniasis. In *CEC*, pages 3052–3059. IEEE Press, 2009.

[5] T. Ghetiu, F. A. C. Polack, and J. Bown. Argument driven validation of computer simulations – a necessity rather than an option. In *VALID*, pages 1–4. IEEE, 2010.

[6] M. Heiner and D. Gilbert. How might Petri nets enhance your systems biology toolkit. In *Petri Nets*, volume 6709 of *LNCS*, pages 17–37. Springer, 2011.

[7] L. Hillah, F. Kordon, L. Petrucci, and N. Trèves. Building an API for ISO/IEC 15909, based on model engineering techniques. Technical report, MeFoSyLoMa, 2005. www.mefosyloma.fr/pdf/ISO-IEC-15909/pnNewsLetter6.pdf.

[8] L. M. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Trèves. The Petri Net Markup Language and ISO/IEC 15909-2. In *Practical Use of Coloured Petri Nets and the CPN Tools*. Aarhus University, 2009. http://cs.au.dk/cpnets/workshops/2009/.

[9] L.-M. Hillah, F. Kordon, L. Petrucci, and N. Trèves. PNML framework: An extendable reference implementation of the Petri Net Markup Language. In *Petri Nets*, volume 6128 of *LNCS*, pages 318 – 327. Springer, 2010.

[10] Software and Systems Engineering – High-level Petri Nets Part 2: Transfer Format. International Standard ISO/IEC 15909, 2005. WD 15909-2:2005(E): www.petrinets.info/docs/ISO-IEC15909-2.WD.V0.9.0.pdf.

[11] T. Katsube, Y. Yano, T. Wajima, Y. Yamano, and M. Takano. Pharmacokinetic/pharmacodynamic modeling and simulation to determine effective dosage regimens for doripenem. *Journal of Pharmaceutical Sciences*, 99(5):2483–91, 2010.

[12] H. Kitano, A. Funahashi, Y. Matsuoka, and K. Oda. Using process diagrams for the graphical representation of biological networks. *Nature Biotechnology*, 23(8):961–6, 2005.

[13] K. W. Kohn and M. I. Aladjem. Circuit diagrams for biological networks. *Molecular Systems Biology*, 2, 2006. online: doi: 10.1038/msb4100044.

[14] D. S. Kolovos, L. M. Rose, N. D. Matragkas, R. F. Paige, F. A. C. Polack, and K. J. Fernandes. Constructing and navigating non-invasive model decorations. In *ICMT*, volume 6142 of *LNCS*, pages 138–152. Springer, 2010.

[15] C. Y. Laporte, M. Doucet, P. Bourque1, and Y. Belkébir. Utilization of a set of software engineering roles for a multinational organization. In *PROFES*, volume 4589 of *LNCS*. Springer, 2007.

[16] F. A. C. Polack, P. S. Andrews, T. Ghetiu, M. Read, S. Stepney, J. Timmis, and A. T. Sampson. Reflections on the simulation of complex systems for science. In *ICECCS*, pages 276–285. IEEE Press, 2010.

[17] F. A. C. Polack, A. Droop, P. Garnett, T. Ghetiu, and S. Stepney. Simulation validation: exploring the suitability of a simulation of cell division and differentiation in the prostate. In *CoSMoS Workshop*, pages 113 – 133. Luniver Press, 2011.

[18] M. Read, P. S. Andrews, J. Timmis, and V. Kumar. A domain model of Experimental Autoimmune Encephalomyelitis. In *CoSMoS Workshop*, pages 9–44. Luniver Press, 2009.

[19] M. Read, P. S. Andrews, J. Timmis, and V. Kumar. Using UML to model EAE and its regulatory network. In *ICARIS*, volume 5666 of *LNCS*. Springer, 2009.