JOT
JOURNAL OF
OBJECT TECHNOLOGY

On developing and validating dynamic systems: simulation engineering

Fiona Polack* **and Kieran Alden†**
*Keele University, UK
†University of York, UK

ABSTRACT Engineering dynamic systems – systems where the behaviour is the dominant characteristic – has some fundamental differences and challenges that are often neglected in model-based software engineering. In engineering simulations, observable system behaviour is built up from the behaviour of low-level components; such simulations are used as research tools in biological and social systems research. A necessary part of simulation engineering is to show that the simulation is fit for its intended purpose, which is easier at the model level than the code level, because the models can be presented in a readable form that domain experts can access.

We explore issues in the use of model transformation for simulation development, using a published Java Mason simulator, created using the CoSMoS approach from UML-style state diagrams. We succeed in recreating part of the class structure of the code by manual transformation, but also expose many issues to be overcome before an automated transformation can be developed, in terms of what needs to be modelled to enable transformation, and how and when design decisions are taken. We identify that a transformation of a behavioural design into an OO simulation also needs a means to capture the low-level simulation and visualisation mechanisms, and a means to capture the design of the behavioural and data aspects of simulation experimentation.

KEYWORDS Complex systems, Simulation, Validation, Model driven engineering

1. Introduction

Software engineering and software modelling has been dominated by approaches to modelling data structure since the advent of relational databases, a focus reinforced by the dominance of object-oriented (OO) programming. Engineering dynamic systems – systems where the behaviour of the system is the dominant characteristic – requires a different approach to modelling and validation. There is a significant body of research and practice on the behaviour-based engineering of safety critical systems (e.g. working from Simulink or similar designs). This work depends on the equivalence of behavioural models (e.g. state diagrams) and mathematical models of dynamical systems: in effect, a special case of behaviour-dominated systems, with the strong formal underpinning necessary to support critical-systems development.

Our systems of interest are complex systems simulations, mostly agent-based simulations, engineered to be fit for the purpose of generating and analysing specific hypotheses about real-world systems. The engineering is not, and cannot be, exact, because the systems being engineered are analogies of real systems that are not, themselves, well-understood or understandable.

There are many reasons for wanting to engineer research simulations. For instance, a research simulation supports:

– entirely repeatable experiments;

JOT reference format:
Fiona Polack and Kieran Alden. *On developing and validating dynamic systems: simulation engineering*. Journal of Object Technology. Vol. 19, No. 4, 2020. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2020.19.4.a1
JOT reference format:
Fiona Polack and Kieran Alden. *On developing and validating dynamic systems: simulation engineering*. Journal of Object Technology. Vol. 19, No. 4, 2020. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2020.19.4.a1

– systematic modification of experiments;
– unlimited data generation.

Furthermore, a research simulation does not require live animals or human subjects, so avoids many ethical and privacy concerns.

There are well-known approaches available for developing behavioural systems (see (Polack et al. 2009) for an early discussion of approaches). The CoSMoS process (Stepney & Polack 2018) provides a lifecycle and techniques for the engineering a demonstrably fit-for-purpose complex systems simulation, and has been used for biological and robotic systems simulations. CoSMoS advocates, but does not demonstrate, the use of model-driven engineering to reduce the uncertainty of capturing a simulation design in simulation code. There has been some underpinning research on MDE for behavioural modelling (e.g. (Polack 2012)), but nothing on model transformation either at the specification level or for code generation from models.

To develop an approach that uses model transformation to derive an agent-based simulation from complex-system behavioural models, a first step is to understand how the information in the design process maps to the implementation of a simulation. In this paper, the proposition is to start from published behavioural specifications[1], and to derive a class model that would be a starting point for an OO agent-based system implementation.

The case study uses the documented development of a Java Mason agent simulation, following the CoSMoS approach: Alden's PPSim[2] (Alden 2012; Alden et al. 2012). The design uses behavioural UML-style models to capture first domain behaviour and then a platform model which is (a) demonstrably derived from the domain model and (b) amenable to the development of the software platform. The model has been fully validated with domain experts, as appropriate for the defined simulation purpose, namely the exploration of hypotheses concerning the timing and implicated behaviours of Peyer's patch formation in the gut of a mouse embryo. The implemented agent simulator has been extensively studied, both as the origin of new insights in its domain (Alden et al. 2012) and as a case study in the engineering of fit-for-purpose simulations.

A complex system has behaviour at many scales. From the perspective of an outside observer, there is emergent, system-level behaviour. The observed behaviour is a consequence of lower level, smaller-scale systems interacting. In engineering a complex system simulation, care must be taken to to determine the scales and abstractions at which to represent lower-level behaviours, and at which to observe and measure the emergent higher-level behaviours. We cannot include everything in the simulation: (a) we could not execute such a large simulation; (b) we do not know about everything; (c) a simulation containing everything would be as complex as the original and thus of limited help to the researchers – in general, it takes a universe with the same starting conditions and evolution as ours to simulate exactly any complex system (Polack et al. 2010). There is no method or short-cut to guide the identification and representation of implicated behaviours; there is only domain expert judgement, engineering judgement, and trial and error. The resulting simulation will be a simplification of all the systems in the real world that (might) be part of or interact with our designated system of interest; furthermore, anything represented directly in the simulation not only represents its real equivalent, but also acts as a surrogate for things not expressed in the simulation.

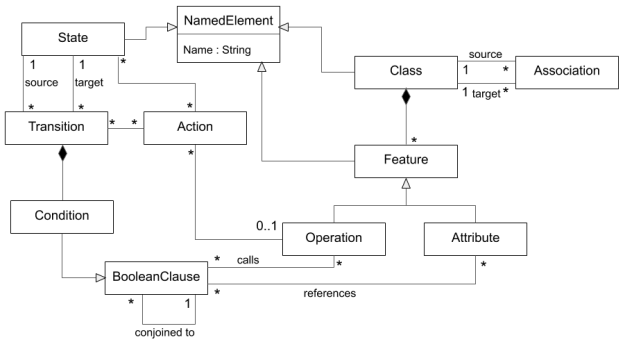

Figure 1 Part of the metamodel for state diagrams and class models in UML, based on UML 2.x and MOF metamodelling. The BooleanClause class is shown as referencing the Attribute class and calling the Operation class: in practice, Booleans reference the name or value of an object slot, and call the operations of an object.

2. Establishing a basis for simulation by model transformation

Simulation engineering focuses on low-level behaviours. Variants of state diagrams and Petri nets are widely used, and can be combined to good effect (Polack 2012).

Agent platforms come in many forms, but Java-based platforms offer a well-supported, flexible programming basis, with good visuals and data collection capabilities. We wish to create a development approach using model transformation, but to fully exploit an OO platform, it is necessary to create a class model from the behavioural design.

UML has an established abstract syntax that links behavioural and class concepts[3]. The parts of a UML metamodel that relate state diagrams and class diagrams are summarised in Fig. 1 (we do not include the metamodel constraints: for instance, a full metamodel might include a constraint that the transition conditions from any state are complete – i.e. that the

[1] Anonymous reviewers wondered why the transformation would start from "high-level state machines", and why we would not develop a class diagram in parallel: the simple answer is that the diagrams presented here are the design models developed, some six years earlier, by the project, and are typical of design models created in related projects that the authors were involved with. The diagrams have to be understandable to immunologists, in order to allow crucial review and checking activities. There is no original class diagram, as immunology comprises cells not objects, and the key feature of a cell is its behaviour.

[2] www.kennedy.ox.ac.uk/technologies/resources/ppsim-peyers-patch-development-simulator created by Kieran Alden.

[3] OMG's UML metamodels: https://www.omg.org/spec/UML/About-UML/

conditions (guards) cover all possible situations). The state diagram concepts capture the states of interest in the lifetime of an object of a class, and transitions between states, whilst the class diagram concepts capture the data structure in terms of attributes and operations that underpin OO models and implementation. Using a common metamodel allows us to confidently relate concepts between diagrams, and establishes the basis of model transformation (Czarnecki & Helsen 2006). The key concept correspondences include the following.

- The state diagram references one class, defining permitted behaviours of its objects; the states of the state diagram are defined over the values of attributes of that class; a state has duration.
- The condition referenced by a transition is a Boolean clause; the condition must be true for the transition to occur; a transition is instantaneous.
- The literals in a Boolean clause may be (values of) attributes of (objects of) the class or of any class which is the target of an association from the class.
- A Boolean clause may imply execution of operations of the class or any class which is the target of an association from the class.
- An action (entry, exit, or called whilst an object is in a particular state) is a call to one or more class operations.

For agent simulation engineering, a state diagram represents agent behaviours, and a class diagram models agent types, operations and interactions. In general, agents are not synonymous with objects, but in practice, and because the development targets an OO implementation medium, the abstract syntax and semantics of UML diagrams can be considered consistent with the agent design.

2.1. State diagram concrete syntax

The concrete syntax used for the UML-style state diagrams (Figures 2 and 3) represents object creation by an arrow (transition) from the solid black dot to a state; object termination is an arrow from a state to a target symbol (there are no object terminations here). States are soft boxes containing the state name and labelled actions that take place on entry, on exit or during the period that an object is in the state; these actions can change the value of attributes of the object, or reference linked objects, etc. A transition is an arrow labelled with the condition on the transition; a transition is triggered either when an object's values become inconsistent with its current state, or when the condition on the transition becomes true.

3. Validating behaviour designs

The complex-systems simulation development scenario is one that even critical systems engineers are not familiar with: rather than engineering a system that minimises uncertainty, risk or hazards, we are developing a software simulation that seeks to faithfully replicate the uncertainty of an incompletely-understood reality. In the simulation engineering context, simulation validity is not binary, but is an argument that captures many assumptions and uncertainties – an argument that must be revisited if the domain understanding, the design, or the simulation purpose is modified. CoSMoS recommends capturing formal or informal arguments of fitness for purpose at each stage of development (Ghetiu et al. 2009; Alden et al. 2011; Polack 2015; Stepney & Polack 2018), capturing the essential basis of perceived trustworthiness of the system for its designated purpose.

The process of validation includes conventional software testing and validation, and also trial-and-error tuning of the simulation so that the desired emergent behaviour can be shown to arise from behaviours and parameterisation that is an acceptable match to the real system. In the case study, PPSim, significant effort went into (a) modelling the relevant parts of the biological domain and arguing its validity; (b) reviewing the models and arguments with domain experts, to confirm their appropriateness to specific experiments (Alden 2012). Compared to conventional software engineering, the published fitness for purpose arguments are neither complete nor sufficient: they only establish conditional validity. However, they present the basis on which trust is established in the simulation, which is the best we can hope for in complex systems simulation.

There are many reasons for wishing to use model transformation to derive simulation code from validated, or at least arguably fit for purpose, designs:

- transformation reduces the risk of coding errors – any errors are in the transformation rules, and are thus systematic (and perhaps therefore easier to spot);
- transformation is repeatable – the same rules applied to the same diagram produces the same code; by extension, the same rules applied to an amended diagram would produce appropriately amended code;
- transformation enables quality simulation creation without significant software engineering skills and insights – the use of transformation (once the transformations have been written) frees the expert software engineer to focus on solving challenging software engineering problems such as finding representations that optimise computational efficiency with understandability (of the models) and usability (of the simulator and its results).

A validation step on design models, which is essential for model transformation, is to check that the models conform to their metamodel. In this case, the state diagrams can be shown to conform to the metamodel in Fig. 2.1. In addition, the state diagrams need to be logically consistent, for example:

1. all literals or value expressed in every transition condition should be consistent with the definition of the state from which the transition is made;
2. the transitions from a state should form a logically complete set;
3. the concepts referred to by a condition or action are features or values of the object, class or linked (associated) objects (classes).

However, conformance checking is not sufficient. It is easy to create a conformant diagram that is not a valid representation of

the domain. In the PPSim case study, for instance, the challenges of simplifying and translating biological behaviour, states and controls into computational language mean that model structure and semantics are sometimes shown to be inconsistent with the domain – we are still discovering issues with the relatively simple state diagrams that describe PPSim cells. A further motivation for using model transformation to implement an agent simulation from design diagrams is to enable efficient code modification when errors are identified in the design.

4. The PPSim Design: state diagram models of cells

The original PPSim design was developed by K. Alden (Alden 2012), working with the CoSMoS team and domain experts led by Viega-Fernandes (see e.g. (Veiga-Fernandes et al. 2007)).

The diagrammatic design comprises state diagrams for each of three implicated cell types, known as LT_o, LT_i, $LT_{i(n)}$, as well as an activity diagram (which is outside the present scope) summarising the cyclic interactions of the cells and the environment. There is a text discussion of the representation of each concept from the domain, and a comprehensive argument that the model is fit for purpose (Alden 2012). In order to present a coherent view of the case study design, the labelling of the state diagrams has been simplified to omit features only used in experimentation with cell division and time-outs.

The three types of cell are all located in a continuous space representing the mouse gut; LT_o cells stick to the gut wall, whilst LT_i and $LT_{i(n)}$ cells move within the dominant direction of flow. LT_o and LT_i cells can bind a "RET Ligand". A bound LT_o expresses chemokine which results in chemokine gradients; LT_i cells can detect the local chemokine and may perform chemotaxis.

The notes accompanying the original state diagrams (Alden 2012) explain features of motion, contact and binding:

- contact means that cells are touching, defined as cell centres being separated by no more than half the diameter of each cell type;
- when cells are in contact, a bind occurs if a generated random value is smaller than the bind limit for the cell determined by calibration, referred to as χ;
- chemokine level is determined by distance from a LT_o, according to a set of research-based diffusion curves[4];
- adhesion (in)sufficiency is defined via a probability of prolonged cellular contact, based on a calibrated parameter, adhesionSlope[5];
- each LT_i and $LT_{i(n)}$ cell has a speed drawn from a Gaussian distribution, calibrated to match observed cell speeds.

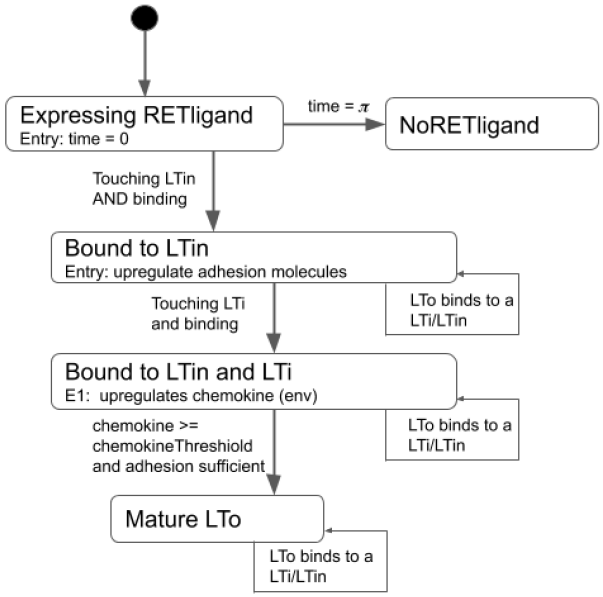

Figure 2 State diagram of LT_o cell, based on (Alden 2012), with addition of non-state-changing transitions recording each binding (these are implicit in the original model). Notations and meaning: see Section 2.1.

The model captures cell-level behaviours; the emergent behaviour that should arise is the formation of clusters. In line with conventions in complex systems modelling, there is nothing in the model that requires or programs the formation of a cluster: a cluster arises as a consequence of the behaviour of many cells, and can be tuned by adjusting cell characteristics, thresholds, creation events, and parameters of the system (referred to as calibration and sensitivity analysis) (Stepney & Polack 2018; Read 2011; Alden et al. 2016).

4.1. LT_o Cells

Peyer's patch formation takes place around an active LT_o cell, a cell type that can bind a "RET Ligand". In the simulation, a LT_o cell is created *in situ* and does not move; other cells may come in to contact or bind the LT_o cell (Alden 2012).

Fig. 2 shows that a LT_o may be in a state that allows Peyer's patch development (expressing RETligand) or not; there is a time-out for a cell in the that state. The state of the LT_o cell changes as other cells make contact and bind, so, when the LT_o cell binds to its first $LT_{i(n)}$ cell, the transition labelled Touching LTin AND binding takes place, changing the cell state to Bound

[4] (Alden 2012) states: *The initial curve is tight, calculated through using the initial chemokine expression level assigned to parameter initialChemokineExpressionLevel. This models expression over a limited distance, but one which strengthens as distance to the LTo reduces. With each stable contact between an LTo and LTi cell, the curve is relaxed by adjusting the parameter increaseChemoExpression, representing an increase in expression. With this increase, diffusion affects LTi cells over a greater distance. This expression increases until a maximum level of expression is reached, set by parameter maxChemokineExpressionLevel.*

[5] (Alden 2012) states: *Each LTo cell has the same initial adhesion factor expression level (set by parameter initialAdhesion). With each stable contact, the level of adhesion factor expression increases (parameter: adhesionIncrement). This increases the probability that a cell remains in the vicinity of the LTo cell for a prolonged period. This probability increases until a threshold is reached (parameter: maxProbabilityOfAdhesion)... there is a chance that an LTin/LTi cell may move away from the forming primordial patch.*

to LTin. The next state change occurs when the LT_o cell binds to a LT_i cell, transitioning the cell to the state Bound to LTin and LTi; this initiates chemokine upregulation. The final transition occurs when a sufficient strength of adhesion (binding) and chemokine is reached, and is described as "mature" (Alden 2012) – in simple terms, the LT_o cell is now firmly bound to a cell cluster.

4.2. LT_i and $LT_{i(n)}$ Cells

LT_i and $LT_{i(n)}$ cells share many characteristics (at least in simulation), differing only in that a $LT_{i(n)}$ cell is not responsive to chemokine. The state diagrams in Fig. 3 have been adapted from the originals principally to apply UML conventions and to avoid confusion across transition conditions, states and events. Movement, which is based on a random walk, is modelled as an action during a state. The form of movement is modelled as different when mediated by chemokine reception, when the cell is in contact with a LT_i, and when bound to a LT_o cell.

4.3. Agent Simulation Environment

A platform such as Java Mason provides a customisable environment for agent interaction. Whilst the environment is not a within the scope of the model transformation considered here, it is useful to note some of the features provided, based on (Alden 2012).

- The simulation has a standard time step, and every agent is visited and updated in each step – PPSim has a detailed timing model which relates biological time to time steps, but that is not in scope here.
- There is an underlying spatial model, and agents have a location in the space – PPSim uses a continuous 2D space to represent a 3D tube.
- Experimentation may require measurement, encoded as data outputs.
- A simulation can be run in visual or non-visual mode. In the latter, execution is faster and can be used to gather large sets of output data for experimental use.

5. Creating a manual transformation

As a starting point for considering model transformation, this section describes the manual derivation of a class diagram from state diagrams. The steps exploit the metamodel relationships between concepts, Fig. 1.

1. Use meta-information (which diagrams exist) to identify classes, and propose generalisations.
2. Represent the named states for each class using attributes.
3. Represent references to (objects of, other) classes in transition conditions, using associations.
4. Systematically consider conditions and actions; determine attributes of the class or other classes.
5. Systematically review conditions, actions, and class features already identified, deriving class operations needed to update attribute values and enact the behaviours determined by actions.

Note that any step might identify other features as a side-effect of its main focus. In general, derivation may require a design decision, which might consider information not available from the state diagrams. In applying the steps, we use the state diagrams and explanatory notes from (Alden 2012), outlined in section 4.

5.1. Introductory remarks

Sections 5 and 6 attempt to convey a process of systematic, rule-based manual transformation in sufficient detail to support repeatability. Inevitably, this leads to dense examples that may be hard for non-specialists to follow. Both authors are software engineers; we have tried to make the terminology accessible to non-specialists (like ourselves), whilst staying faithful to the published design. We had the option to turn the diagrams into nicer software models that map more cleanly into class diagrams, or to simplify the diagram labelling, but chose to describe a real starting point and a real end point in order to explore whether it is possible and realistic to attempt transformation from real domain and design models to real OO classes. The main points that arise are summarised in the discussion and conclusions, Sections 7 and 8; the manual transformation has been very time-consuming, and it is clear that an ideal development would have created a different set of starting models, but the principle of using model management to support fit-for-purpose simulation development remains.

5.2. Step 1: Identify Classes and Generalisations

The set of models is the meta-information used to determine the set of classes, one for each state diagram:

- LTo class
- LTi class
- LTin) class

However, mobile cells share many behavioural characteristics, and all three state diagrams relate to types of cell that share attributes such as a location, and behaviours related to touching other cells. We can therefore deduce generalisations: Cell for the common features of all cells and MobileCell for the common features of LT_i and $LT_{i(n)}$.

5.3. Step 2: Add state attributes

There are several ways to represent state-diagram states in a class. Each state can be represented as a separate Boolean attribute: the attribute is *true* when an object is in this state. This representation requires conditions: (a) at most one state attribute can be true for an object at any time; and (b) a defined partial order expressing permitted state changes. Alternatively, a single state attribute with an ordered type, such as Integer, avoids the need for both conditions.

Here, a compromise solution, which is more readable – an important consideration when models are validated by domain experts as well as software engineers – but requires an ordering constraint, is to use enumerated types, represented in UML as an «enum» type class (Fig. 4).

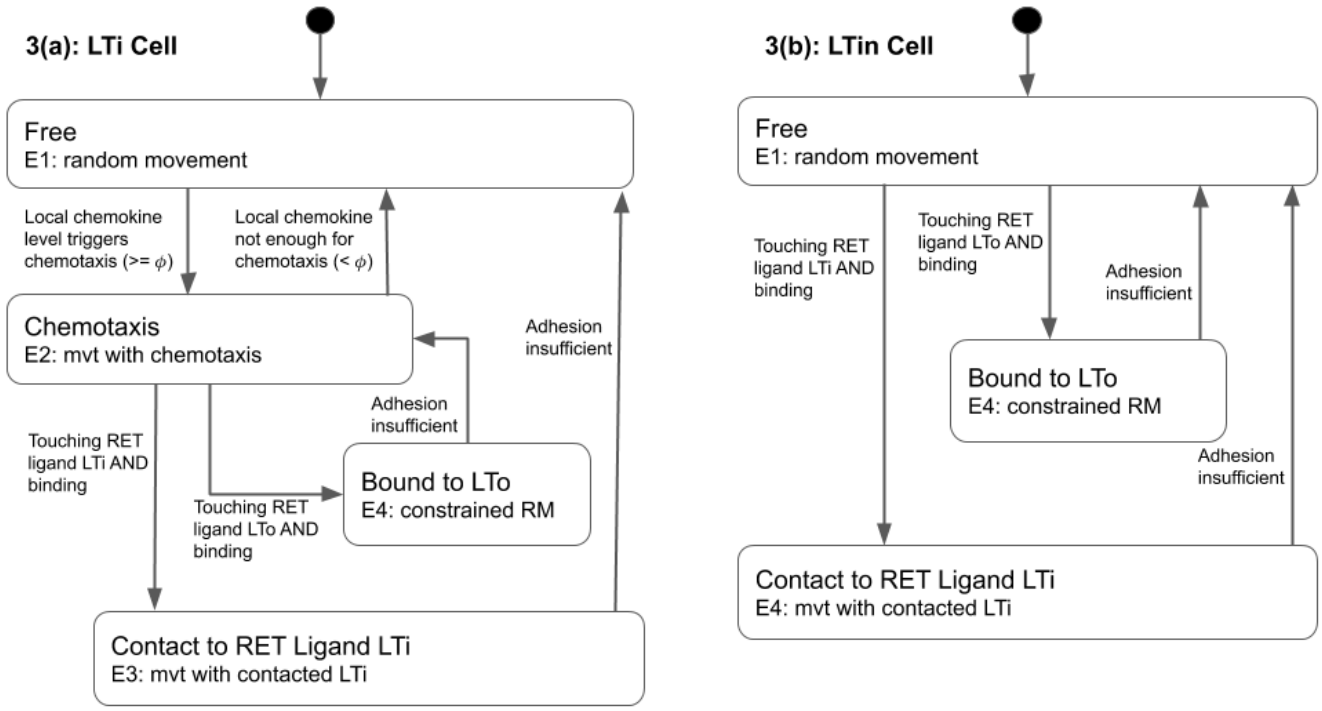

Figure 3 State diagram models of LT_I (Fig. 2(a)) and $LT_{i(n)}$ (Fig. 2(b)) cell, based on (Alden 2012). Notations and meaning: see Section 2.1.

5.4. Step 3: Identify associations implied by conditions

A UML state diagram captures the life-cycle of objects of one class but conditions can reference properties of other objects or classes. To identify associations, we systematically review every condition on each state diagram.

The LT_o transition conditions record contact and binding with $LT_{i(n)}$ and/or LT_i cells (Fig. 2, above), implying associations from the LTo class to each type of mobile cell, or to the parent class, MobileCell.

In terms of multiplicity, the general association is a 1:m association: one LT_o may bind any number of mobile cells; each mobile cell can bind to at most one LT_o. The notes accompanying the state diagram indicate that factors such as binding are mediated by the number of contacts or bindings, and thus the number of links is important: this can be recorded in an attribute, and/or calculated by running a function over the association links.

There are also two specific bindings that are important to LT_o state changes: the first bind to a $LT_{i(n)}$ and the first bind to a LT_i. These conditions are optional 1:1 associations, and, as such, could be replaced with attributes (Boolean or of the type of the linked cell).

The derivations used are:

- a 1:m association, isBoundTo between LTo and MobileCell;
- attributes LTo.boundLTi and LTo.boundLTin to identify first bindings;
- an attribute to facilitate calculation of binding, LTo.countBinds

Both the LT_i and $LT_{i(n)}$ state diagrams have transition conditions relating to contact. These conditions exactly imply the inverse relationship. The notes with the state diagram describe chemotaxis of a LTi and the constrained movement of a mobile cell that is bound to a LTo: this implies that the location, and thus the identity, of the bound LTo is important. We propose MobileCell.boundLTo.

The classes, generalisations and association are shown in Fig. 5.

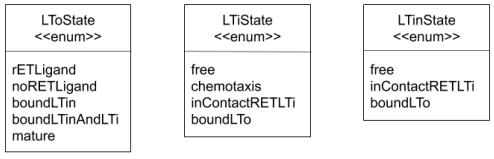

Figure 4 «enum» stereotype defining the states of cells, from Figures 2 and 3.

5.5. Step 4: Identify class attributes

In addition to the state attributes (Section 2), conditions imply attributes; the notes accompanying the state diagrams help to determine some of the design details. To avoid making arbitrary decisions, the types of attributes are descriptive (e.g. adhesion-related attributes have a type, AdhesionType), unless their type is uncontroversial (Integer or Boolean).

The relevant conditions from the state diagrams (Figures 2 are:

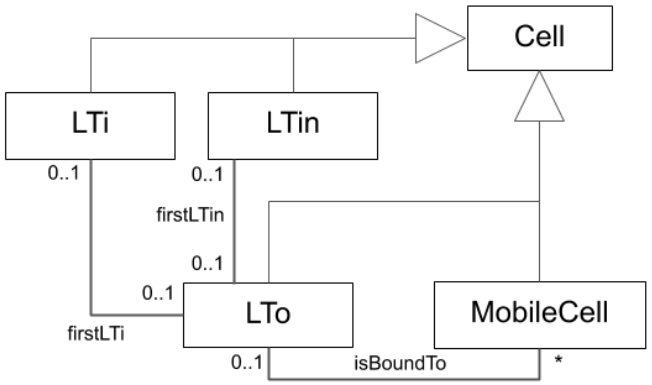

Figure 5 Adding associations to the base class diagram.

- the LT_o condition time = π and the action Entry: time = 0;
- the LT_o condition chemokine = maxChemokine and adhesion sufficient;
- the LT_i conditions Local chemokine level triggers chemotaxis (>= ϕ) and Local chemokine level not enough for chemotaxis (>= ϕ);
- the LT_i and $LT_{i(n)}$ conditions Adhesion sufficient and Adhesion insufficient.

time refers to a local timer on the LT_o cell which can be incremented to a threshold level, π. In (Alden 2012), the notes accompanying the state diagram states that π is a simulation parameter.

For chemokine, the notes accompanying the LT_o state diagram (Alden 2012) describe the derivation of chemokine gradients and associated initial and maximum values: these are simulation parameters set by calibration. Each cell has a chemoExpressionLevel that is calculated from the chemokine curve. A LT_i cell calculates the localChemokineLevel based on the chemoExpressionLevel of its nearest LT_o and their relative locations.

The notes on adhesion explain that cells have a probability of adhesion that increases with duration of contact (Alden 2012); all cells have an adhesion factor expression level, adhesionExpressionLevel that is initially set to 0. There are simulation parameters to establish how the adhesion level is set and incremented.

In summary, the derived attributes are:

- LTo.time = 0
- LTo.chemoExpressionLevel
- LTi.localChemokineLevel
- Cell.adhesionExpressionLevel = 0

5.6. Step 5: Deriving class operations

The scope of operation coverage here is limited; although some aspects of composite operations can be deduced from the notes with the state diagrams, the original design does not include sequence diagrams or equivalent that describe the composite operations in detail.

The class model needs to provide operations that set, get (by passing messages over object links) and adjust the values of attributes, as well as to implement the actions and condition evaluations in the state diagrams. Setters and getters are straightforward, and are usually omitted from UML class diagrams.

For the state attributes on the cell classes, operations need to check the conditions on transitions at each time step, and advance the state when a transition condition is true. The state change operations are:

- LTo.changeState()
- LTi.changeState()
- LTin.changeState()

Turning to the attributes identified in the previous steps, the attribute LTo.time has two implied operations: increment and comparison the counter value to the simulation parameter, π:

- LTo.incrementTime()
- LTo.checkTime()

For adhesion, operations are needed to evaluate adhesion sufficiency.

- LTo.incrementAdhesion()
- MobileCell.calculateAdhesionProbability()

Similarly, operations are needed for chemokine expression, evaluation, and triggering chemotaxis:

- LTo.updateChemoExpressionLevel()
- LTi.calculateLocalChemoLevel()
- LTi.establishChemotaxis()

The state diagram actions also imply operations. The LT_o action Entry: time = 0 has been accounted for by setting the initial value of the time attribute to 0. The action on state, Bound to LTin, Entry: upregulate adhesion molecules, corresponds to the operation derived to incrementAdhesion(). Similarly, the action on the Bound to LTin and LTi state, E1: upregulates chemokine (env), is captured by updateChemoExpressionLevel().

The actions on LT_i and $LT_{i(n)}$ cell states concern the form of motion. From the notes accompanying the original state diagrams (Alden 2012), every cell has a speed drawn at random from a Gaussian distribution, which determines how far it moves in each time-step. The direction of travel in the time-step is determined as follows:

- as a random walk (E1: random movement in Fig. 3), until cell interactions start;
- as a chemotaxis-weighted random walk (E2: mvt with chemotaxis) in which the probability of moving in any direction is related to chemokine strength, calculated from the relative location of a LT_o emitting chemokine;
- as a random walk weighted by the adhesion strength and a probability that a bound cell can move away from the LT_o (E4: constrained RM).

Fig. 3 also includes a possible additional movement, E3: mvt with contacted LTi, where two LT_i cells are in contact (this is not

π : Integer *(time)*
ϕ : ChemokineType
initialChemokineExpressionValue : ChemokineType
maxChemokineExpressionValue : ChemokineType
maxProbabilityOfAdhesion : AdhesionType
adhesionIncrement : AdhesionType

Table 1 Simulation parameters derived from state diagrams and accompanying notes (Alden 2012).

in the original model, which does not distinguish RET-binding to by cell type). This form of movement, and the chemotaxis-weighted random walk, only apply to LT_i cells. The derived operations are:

- MobileCell.calculateSpeed()
- MobileCell.moveRandomWalk()
- MobileCell.constrainedRandomWalk()
- LTi.chemotaxisRandomWalk()
- LTi.contactLTiRandomWalk()

Reviewing the operation derivation, it is apparent that many conditions relate to contact or binding and the calculations of adhesion and bind strength. We therefore derive two further generic operations,

- Cell.calculateContact(Cell,Cell)
- Cell.calculateBind(LTo,MobileCell)

The full set of derived classes, attributes and operations is shown in Fig. 6 and Table 1.

6. Validating the derived model

Manual derivation of the class diagram has carefully checked each step against information in the state diagrams, and accompanying notes (in (Alden 2012, Chapter 2)). As previously noted, manual processes are error-prone: the manual PPSim derivation has been revisited five times, and although the class structure is consistent, the detail of each class differs depending on how each feature of the state diagram and notes is interpreted.

Since the PPSim code exists, there is a more independent validation check, to compare the classes derived manually (the *derived model*) with the structure of the classes in the original PPSim code (the *code model*), which can be extracted by any development environment capable of representing code structure as a UML-style class model[6]. The code model classes, extracted from the PPSim code, include setters and getters, making the image too large to reproduce clearly; Fig. 7 is included for completeness only.

[6] An anonymous reviewer makes the excellent point that it is also important to ensure that the behaviour represented in the models matches the behaviour of the code: this is a part of the simulation validation, typically entailing extensive calibration and sensitivity analysis, and is beyond the scope of this paper exploring the basis for use of model transformation.

The most obvious difference between the derived and code models is that the code only uses one generalisation, `Cells` (sic). This means that many attributes and operations appear in different places to, and have wider scope than in the derived model. Furthermore, it becomes challenging to compare the models class-by-class. Instead, the following addresses the generic concepts in turn: state and time, chemokines and adhesion, movement, contact and binding.

6.1. Derived and code models of state and time

In the code model, `Cells.cellState:int` represents the state of any cell, whereas the derived model has an enumerated state attribute for each cell class. Whilst there are advantages to either design, the representations are conceptually similar.

State change in the derived model is an operation on each cell class; this would allow a consistent implementation in each class, calling the operations needed to execute checks and comparisons, and run actions.

By contrast, state change is not handled systematically in the code model: there are many operations related to the behaviour that causes and arises from state change in `Cells` class, coded from the biological descriptions, rather than from the state diagrams. No single `Cells` operation encodes all state changes or all effects of a state change for any cell type. The relevant Cells operations are:

- `Cells.alterLTiState()`: the condition and effect of a `LTi` binding to a `LTo`;
- `Cells.updateLToState()`: one state change, part of the derived-model operation, LTo.changeState();
- `Cells.updateRLNSState()`: the effect of a non-stromal (i.e. mobile) cell expressing RET-ligand – the platform model does not have any corresponding terminology;
- `Cells.artnRETSignalling()` and `isExpressingRETLigand()`: `LTo` and `LTi` RET signalling, implicated in state changes – encodes an interpretation of the biology rather than the behaviour modelled in the state diagrams;
- `Cells.immatureLToActivation()`: part of the `LTo` life-cycle – *immature* is a biological label for a LT_o that is still undergoing contact and binding.

In the derived model, the only reference to time is the LT_o deactivation time, with derived attribute LTo.time and operations LTo.incrementTime() and LTo.checkTime(). The equivalent attribute in the code model is `LTo.activeTime`. In the code model, checking time is conflated with the effect of exceeding the time limit, in operations `activateRETLigand()` and `removeRETLigand()`.

In the code model, there are three more time-related attributes that record how long a `LTo` cell has been in contact with another cell (`lTinContactStateChangeTimePoint`, `lTiContactStateChangeTimePoint`, `matureLToStateChangeTimePoint`; these are used in calculating bindings (e.g. the operation `LTo.stableContact()`), at a lower level of detail than the state diagrams. In addition, the code model includes `Cells.timeTracked:int` and

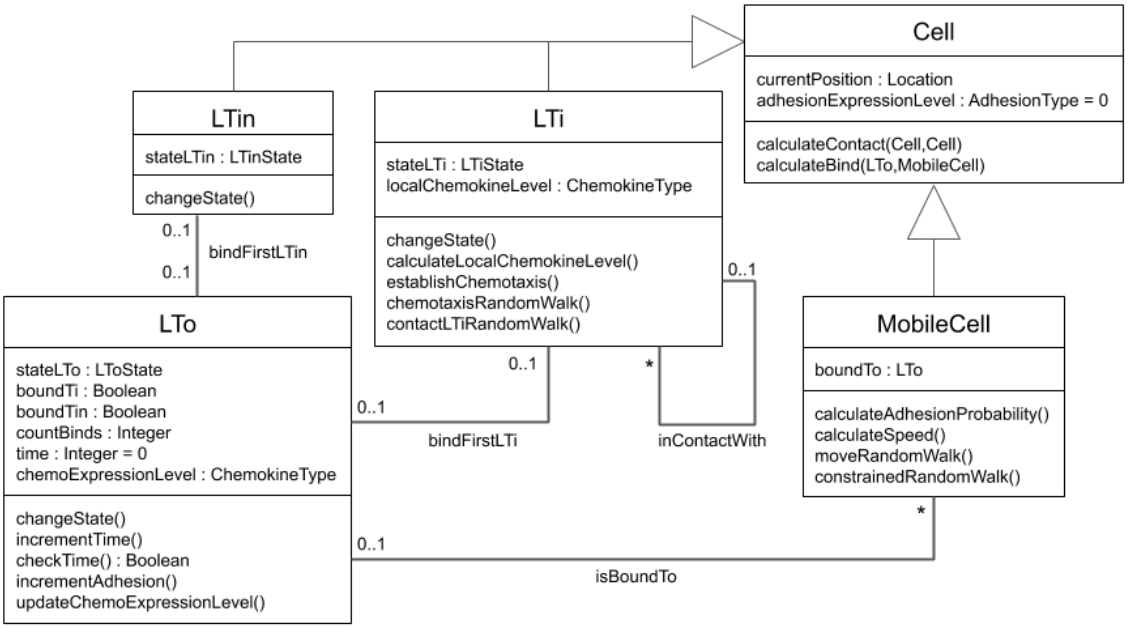

Figure 6 Classes, attributes and operations (omitting setters and getters) derived from state diagrams and accompanying notes (Alden 2012).

`LTi.getTimeTracked()`, which support experimental data collection.

6.2. Derived and code model chemokine representations

In the derived model, the representation of chemokines uses ChemokineType as a placeholder for an abstract data type which, conventionally, would capture the basic operations on as well as the attribute type. The value of LTo.chemoExpressionLevel: ChemokineType is set by LTo.updateChemoExpressionLevel(), as required by the process described in the notes (see Section 5.5, above). Chemokine values are used in determining the behaviour of LT_i cells – LTi.localChemokineLevel is set by LTi.calculateLocalChemoLevel(), which would read LTo.chemoExpressionLevel. The threshold and trigger level are simulation parameters (i.e. not part of the model classes).

In the code model, attributes of the `LTo` cell encode cell-related and system-related parameters: `chemoSigThreshold`, `chemoLinearAdjust`, `startingChemoLinearAdjust` and `endingChemoLinearAdjust:double`. The `LTi` class has no chemokine-related attribute or operation. The model reflects the detailed biological mechanisms, not the state diagrams.

Whilst a direct validation of the derived model is not possible, it appears that there is a similar intention. In relation to the LT_i, the derived model is faithful to the state diagram in Fig. 3, but we note that this is a redrawn version of the platform model, where the effect of chemokine is a change in movement: the code model includes evaluation of chemokine by `LTi` cells in the `Cells` movement operations.

Arguably, the derived model better supports code validation, because it provides clear traceability and separation of concerns – though it has not been determined whether this conceptual clarity would complicate the encoding of low-level behaviours in the OO platform. In software engineering terms, the use of an abstract data type (i.e. a class, with behaviours, that is used as a type) would generally be thought a more appropriate solution than a direct encoding of type-related behaviours.

6.3. Derived and code model adhesion representations

Adhesion applies to all cells, and both the derived and code models position adhesion-related features in the parent class. The derived model again uses an abstract data type, AdhesionType, whereas the code model provides component operations that capture the biology better than they express the state diagrams.

In the derived model, Cell.adhesionExpressionLevel is updated by LTo.incrementAdhesion(); MobileCell.calculateAdhesionProbability() calculates the adhesion probability factor that is then compared with Cell.adhesionExpressionLevel to determine whether adhesion is sufficient to maintain a bind, using Cell.calculateBind().

In the code model, the encoding of low-level biology refers to *VCAM*, from the biochemistry of adhesion (`LTo` attributes `startingVCAMExpressionLevel` and `endingVCAMExpressionLevel`; and `Cells` attribute, `vcamAdhesionEffect`, and operation, `calculateVCAMEffect()`).

As for chemokine, a direct validation is not possible, but again the intention is similar. The presence of the type, `VCAMAdhesionEffect` suggests that an abstract data type ap-

Figure 7 Full details of classes and generalisations in PPSim code

proach is used for part of the code model.

6.4. Derived and code model movement

In the state diagram, Figures 2 and 3, movement changes depend on the influence of chemokines (LT_i cells only) and adhesion. The variants of movement are taken from the notes and labelling of the original state diagrams. In the derived model, movement is a feature of the MobileCell class, which has operations, calculateSpeed(), moveRandomWalk(), and constrainedRandomWalk(). These operations need to access cell locations, binding and adhesion data.

The code model again includes biological and calculation details; all the movement behaviours are on the `Cells` class. The main attribute is `CellSpeed:double`, but there are additional speed attributes, `Cells.cellSpeedScaled:double`, `Cells.cellSpeedSecond:double` and `Cells.trackedVelocity:double` for use in experiments. The operations are at a lower level than those in the domain model, and focus on the effect of movement (to relocate the cell) rather than the movement itself: `performMoveAfterContact()`, `performMove()`, `calculateNewPosition()`, `rollAround()` and `distanceBetweenTwoPoints()`.

The basics of movement (moving a mobile cell to a new location, according to its speed and the constraints imposed by binding) are similar, assuming that the code model's `performMove()`, `performMoveAfterContact()` and `rollAround()` implement the unconstrainted random walk, a random walk constrained by contact, and the movement of a bound cell that cannot break contact, respectively.

The code model has some additional movement-related operations. `Cells.lTiLTinCellCollision()`, `collisionCheck()` and `Cells.avoidCellCollision()` capture the fact that contact and binding are only relevant to cell pairings including a LT_o cell, as well as surrogating for the fact that, in reality, the physics of flow etc., prevent cell collisions. This detailed behaviour is not captured in the state diagrams.

6.5. Derived and code model contact and binding

Contact is defined in the original model as cell centres being within the sum of half their respective diameters (Alden 2012). Binding is calculated for any cells in contact, where one cell is a LT_o, using adhesion level and a generated random number (to maintain stochasticity of binding).

The derived model contact and binding is described in Sections 5.4 and 5.6, above. The LT_o is always the source of the contact or binding. In the code model, the attribute, `LTo.imLToCellContactCount` plays the same role as LTo.countBinds in the derived model. The *im* prefix is, again, a reference to the biological term, *immature*, which covers RET-ligand LT_o cells before maturity. The behaviour associated with immaturity is to record contact and binding to mobile cells, which is captured in Fig. 2 as the ability to undergo a non-state-changing transition in the relevant states. The derived model records two further attributes, `LTo.boundLTi:Boolean` and `LTo.boundLTin:Boolean`, which are implicated in the operatational state changes in Fig. 2. The code model includes an operation, `Cells.findNearestLTo()`, which avoids the need to record which LT_o cell is the current focus of movement.

In the derived model, the parent class has the operations to calculate contact and binding: calculateContact(Cell,Cell) and calculateBind(LTo,Cell): the latter accesses the adhesion attribute and operations of the other classes. In the code model, `Cells.contactedCell` identifies cells that are in contact with `this` cell object.

Again, exact equivalence cannot be shown, but there is common intent in the code and derived models.

7. Discussion

Whilst the approach here shows that a model transformation approach from a CoSMoS-style platform model to OO-based agent simulation is possible in principle, the manual approach highlights many issues. The attempt to validate the classes

derived by a systematic manual transformation against those extracted from the PPSim simulator code reveals that the transformation approach would change the developer approach, restricting creativity in coding solutions but potentially facilitating demonstration of the fitness for purpose of the code. Furthermore, it is likely that transformation could improve code quality and facilitate creation, maintenance and reuse of code.

The manual derivation relies on meta-information such as the existence of state diagrams (representing known classes of cell) and intuition (in generalisation). However, it is arguable that, for code generation, the class structure – the associations and generalisations – could have been ignored or identified in other ways. Generalisation can be retro-fitted, by post-hoc review of features that classes have in common. For associations, it is a well-known issue in OO that coding of object linkage cannot be achieved at class level, so we could potentially ignore association derivation. Instead, the state models could systematically identify *message passing* needed by derived operations: if the platform model included systematic definitions of operations (e.g. sequence diagrams), the required message passing would be clear (but then, so would the required associations).

The detail of the manually-derived classes uses all the features of the state diagram, and applied the conceptual equivalences defined in the metamodel, as well as an intuitive understanding of how conditions and actions reference object features. The manual approach is relatively easy to describe, but draws on the intuition of the modeller; because a condition can access almost any possible value, attribute, object or class, it would be challenging to capture the process as a set of discrete transformation rules. Furthermore, the notes accompanying the original state diagrams were consulted to understand the detail of state diagrams, and this would not be available to an automated transformation. The derivation has not addressed the detail of operations, or even the pre- and post-conditions, which would be an important part of a code implementation.

In this case study, only the state diagrams were used. The original domain model includes an activity diagram, which provides more information concerning operations across classes, and complements the information in the notes. However, what would be most useful for an automated transformation would be addition of sequence diagrams (or equivalent) to describe the key behaviours such as movement, contact and binding. Indeed, it seems that such models are a necessary condition for an automated transformation. This should not be surprising, given that the more early modelling and validation work that is undertaken, the easier it is to accomplish good-quality software development. Tying down the platform specification early in development leaves fewer open options in coding, and makes design decisions easier to record and analyse.

In attempting to validate the derived model against the code model, the effect of open design decisions is very evident. Whilst the effect of different design decisions for recording class states is slight, the derived and code models take very different approaches to underpinning the detailed behaviour of the cells. It is arguable that code derived from the diagrams would be better structured than that manually coded. Transformation would also remove the language problems found in the code-model classes (biological and biochemical terms) – more specifically, transformation would eliminate the possibility of a concept having different names in the models and the code.

It is interesting to note that the detailed notes accompanying the design (Alden 2012) and the PPSim code model both facilitate code-level validation by the domain expert (biologists): the domain expert is reassured that the code captures the calculations that they understand. If a full model transformation were possible, however, there would be no need for the domain expert to re-validate implementation: fitness for purpose argued at the domain and platform model levels would pertain by transformation to the code.

At a more detailed level, it is clear that the code model includes detail that is not captured in the design models – detail that cannot be derived from the design, because it is not there. The code-model classes include attributes and operations used: (a) to record data needed for experimental results; (b) to operate the simulator (stop and start runs); and (c) to support all or some experimental set-ups of the simulation. An automated transformation assumes that only the design models are required to derive code, so further work would be needed, using patterns, templates or other transformation models, to support code generation. There are simulation platforms that encapsulate the simulator code, e.g. through application program interfaces. However, such platforms tend to have significantly reduced flexibility: the power of the OO platforms is their ability to support any behaviour and representation that can be expressed in OO terms. This trade-off between flexibility and convenience is again a common issue in software engineering.

8. Conclusions

A premise of the work presented here is that, for complex systems simulation, model transformation would allow the software engineer to focus on solving real problems rather than just routinely coding. We have shown by manual transformation that the design contains a good part of the information needed to create code. However, it is also the case that behavioural models of algorithms (e.g. sequence diagrams) would be a useful addition, and it is not clear whether the human interpretation used in the manual transformation could be captured as transformation rules.

The attempt to validate the derived class model against the class structure of the PPSim code shows that, at a sufficiently abstract level, the models have similar coverage. However, the manual transformation also revealed different design decisions, and different approaches to supporting the same behaviours. Code created by transformation might have better structure than code created variously from biological detail, platform and domain modules. A clear conclusion is that, by using model transformation of validated models, effort could focus on fitness-for-purpose and trustworthiness at domain and platform levels, rather than on code validation. However, it is also clear that we cannot simply map our manual transformations into a transformation model; we either need richer design models or we need intermediate models to build up to code-level information. We are starting to investigate modelling and model-management

support not only for model-to-code transformations, but also for the process of deriving models of complex systems that will be simulated, and of modelling the experimentation needed to validate and use the simulators. It is also possible that, as simulation continues to grow in use as a tool for researching complex systems, the widely-used Java-based and other OO platforms will be replaced by media more suited to representing systems dominated by behaviour.

The discussion has not addressed the wider aspects of simulation validation. In practice, a research simulation is designed by modelling and coding what is essentially the best-guess design. Because the simulation is complex, the actual behaviour of the simulation is only discovered through running it. In order to align the simulation with the real system, calibration is used: this may result in adjustment of values, or even adjustments in the design. Once a calibrated model produces acceptable behaviour, at least within the intended operational scope, sensitivity analysis is used to ensure that the observed behaviours arise from appropriate parameters and behaviours. This validation activity can also result in adjustments from values through to design details. Better engineering support for such simulation validation could include not only automated model-to-code transformation, but also bidirectional transformation, or round-trip engineering, to ensure that platform models and code are mutually consistent.

Acknowledgments

We would like to acknowledge the two anonymous reviewers of this paper, for their very insightful comments, and their acceptance of our very detailed description of how not to go about designing a system to support the use of model transformation. We sincerely hope that Dr Gogolla will appreciate our attempts at working from behavioural models only, and at validation both of models and our final product, and that he will not be too repelled by two dense sections of pseudo-biology.

References

Alden, K. (2012). *Simulation and statistical techniques to explore lymphoid tissue organogenesis* (Doctoral dissertation, University of York). Retrieved from etheses.whiterose.ac.uk/3220/

Alden, K., Andrews, P., Timmis, J., Veiga-Fernandes, H., & Coles., M. C. (2011). Towards argument-driven validation of an in-silico model of immune tissue organogenesis. In *Icaris* (Vol. 6825, pp. 66–70). Springer. doi: doi.org/10.1007/978-3-642-22371-6_7

Alden, K., Timmis, J., Andrews, P. S., Veiga-Fernandes, H., & Coles, M. C. (2012). Pairing experimentation and computational modelling to understand the role of tissue inducer cells in the development of lymphoid organs. *Frontiers in Immunology*, *3*(172).

Alden, K., Timmis, J., Andrews, P. S., Veiga-Fernandes, H., & Coles, M. C. (2016). Extending and applying Spartan to perform temporal sensitivity analyses for predicting changes in influential biological pathways in computational models. *IEEE Trans. Comp. Bio.*, *14*(2), 431–422.

Czarnecki, K., & Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, *45*(3), 621–645. doi: doi.org/10.1147/sj.453.0621

Ghetiu, T., Alexander, R. D., Andrews, P. S., Polack, F. A. C., & Bown, J. (2009). Equivalence arguments for complex systems simulations - a case-study. In *Complex systems simulation and modelling workshop* (pp. 101–129). Luniver Press. (ISBN: 978-1-905986-32-3)

Polack, F. A. C. (2012). Choosing and adapting design notations in the principled development of complex systems simulations for research. In *Modelling the physical world at models.* ACM Digitial Library. doi: doi.org/10.1145/2491617.2491623

Polack, F. A. C. (2015). Filling gaps in simulation of complex systems: the background and motivation for CoSMoS. *Natural Computing*, *14*(1), 49–62. doi: 10.1007/s11047-014-9462-5

Polack, F. A. C., Andrews, P. S., Ghetiu, T., Read, M., Stepney, S., Timmis, J., & Sampson, A. T. (2010). Reflections on the simulation of complex systems for science. In *Iceccs* (pp. 276–285). IEEE Press. doi: 10.1109/ICECCS.2010.48

Polack, F. A. C., Andrews, P. S., & Sampson, A. T. (2009). The engineering of concurrent simulations of complex systems. In *Cec* (pp. 217–224). IEEE Press. doi: 10.1109/CEC.2009.4982951

Read, M. N. (2011). *Statistical and modelling techniques to build confidence in the investigation of immunology through agent-based simulation* (Doctoral dissertation, University of York). Retrieved from /etheses.whiterose.ac.uk/id/eprint/2174

Stepney, S., & Polack, F. A. C. (2018). *Engineering simulations as scientific instruments: A pattern language*. Springer. doi: 10.1007/978-3-030-01938-9

Veiga-Fernandes, H., Coles, M., Foster, K., Foster, K. E., Patel, A., Williams, A., … Kioussiset, D. (2007). Tyrosine kinase receptor RET is a key regulator of Peyer's Patch organogenesis. *Nature*, *446*, 547 – 551. doi: doi.org/10.1038/nature05597

About the authors

Fiona Polack is Professor of Software Engineering at Keele University, and a former member of the York Computational Immunology Lab at University of York. While at York, she was a member of the software engineering team, now led by Professor Dimitris Kolovos, responsible for the development of the Eclipse Epsilon model management tool suite, co-supervising many model management PhDs in the group.is a professor at Keele University (UK). She works on software engineering of demonstrably fit-for-purpose simulations of complex systems, building on the work of the the CoSMoS Project (EP/E049419/1). You can contact him at f.a.c.polack@keele.ac.uk or visit https://www.keele.ac.uk/scm/staff/fionapolack/.

Kieran Alden was a post-doctoral researcher in the University of York, Department of Electronic Engineering, having completed

his PhD in the York Computational Immunology Lab; he has co-supervised many simulation projects in both immunology and robotics. His work includes open-spource tool support for simulation including calibration, sensitivity analysis and statistical analysis (Spartan, Robospartan, Aspasia). Dr Alden is now Lead Data Scientist for Vianet Group plc. You can contact him at kieran.alden@gmail.com or visit https://www.kieranalden.info/.