On developing and validating dynamic systems: simulation engineering

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

ABSTRACT Dynamic systems, where the behaviour is the dominant characteristic, pose engineering challenges that are often neglected in model-based software engineering. However, supporting simulation development from design models is important in demonstrating that a simulator and simulation experiments are fit for their intended purpose. In engineering agent-based simulations, observable system behaviour is built up from the behaviour of low-level components; such simulations are used as research tools in (for instance) biological systems research. We have found that domain experts can validate diagrammatic models of behaviour and accompanying text, but we need model-based software development, and ideally automatable model transformation, to maintain fitness for purpose into code and experimentation.
We present an exploration of behavioural model transformation, devising and applying manual transformation guidelines to an existing, published Java Mason simulator, created using the CoSMoS approach and UML-style state diagrams. We succeed in recreating part of the class structure of the code, but also expose many issues to be overcome, in terms of what needs to be modelled to enable transformation, and how and when design decisions are taken and documented. We also identify the need to generate the creation of low-level simulation, visualisation, and data capture mechanisms, and a means to design and encode 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 ever since the advent of relational databases, a focus reinforced by the dominance of object-oriented (OO) programming and ubiquitous class modelling. Model driven engineering (MDE), based on models created in languages defined by metamodels, provides a consistent, repeatable approach to deriving OO programs from class models. However, engineering dynamic systems (systems where behaviour is the dominant characteristic) requires a different approach to modelling and validation; whilst model driven approaches exist, they are not universally applicable.

JOT reference format:
Fiona Polack and Kieran Alden. *On developing and validating dynamic systems: simulation engineering*. Journal of Object Technology. Vol. 19, No. 3, 2020. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2020.19.3.a6

The dynamic systems that we engineer are agent-based simulations of complex systems, developed in collaboration with laboratory scientists, and used to generate and explore hypotheses of interest to the scientists. In this paper, we use the design of one such simulation as a case study; we use the published design of the simulator to explore whether it might be possible to use model transformation approaches to derive a code structure. The original simulator was created on the Java Mason OO agent simulation platform, with code hand-crafted with reference to the design models that had been created with and signed-off by the laboratory scientists.

To explore potential model driven engineering, we start, here, by exploring a systematic but manual derivation of a class diagram suitable for model transformation. We compare the results with the class structure of the actual simulator, enabling a better understanding of how to achieve our ultimate goal of model-driven simulation engineering.

1.1. Background

There is a significant body of research and practice on the behaviour-based model-driven engineering of safety critical systems (e.g. working from Simulink or similar designs). This work supports code generation, but depends on the equivalence of behavioural models (e.g. state diagrams) and mathematical models of dynamical systems. Such critical systems engineering demonstrates that formally-correct programs can be derived from behavioural diagrams, but does so by imposing strict constraints on what can be modelled. It is, in effect, a special case of behaviour-dominated systems, with the strong formal underpinning necessary to support critical-systems development. However, when creating research simulations of complex 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 of complex systems. For instance, a research simulation supports:

– entirely repeatable experiments;
– 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.

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 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. 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 is a simplification of all the systems in the real world that might be part of or interact with the designated system of interest. Furthermore, anything represented directly in the research simulation not only represents its real equivalent, but also acts as a surrogate for things not expressed explicitly in the simulation.

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 demonstrably fit-for-purpose complex system simulations, which have been applied in the development of biological and robotic systems simulations. CoSMoS advocates MDE, but CoSMoS is not prescriptive, so it does not have inherent languages on which to base MDE support. There is limited research on MDE for behavioural modelling (e.g. (Polack 2012)), but no simulation development has yet attempted to use it.

We focus here on the CoSMoS-style simulation projects exploring largely cell-level biological systems implicated in immune responses (Alden et al. 2012; Moore et al. 2013; Greaves et al. 2013; Read 2011; Williams et al. 2013). These developments have used UML-style modelling, but no modelling tools, and code has been hand-crafted.

Agent simulation platforms come in many forms, but Java-based platforms offer a well-supported, flexible programming environment, with good visuals and data collection capabilities. A key practical issue is that the behavioural modelling used to explore the domain and prepare for development does not map directly into OO-based agent simulation platforms selected by developers. A first step in employing model transformation targeting agent-based simulation platforms is to understand how the information in the design and documentation maps to the implemented simulation.

The case study here is the documented development of a Java Mason agent simulation, developed using the CoSMoS approach: Alden's PPSim[1] (Alden 2012; Alden et al. 2012). The design uses UML-style state diagrams to capture the domain behaviour, and then reworks the biological behaviours into computational behaviour designs, also represented in state diagrams. All transformation is manual. The documentation records fitness for purpose arguments that give assurance that the computational design is (a) demonstrably derived from the domain model, and (b) appropriate for the development of the software platform. The model has been fully validated with the collaborating domain experts, and demonstrated to be appropriate for the defined simulation purpose — the exploration of hypotheses concerning the timing and implicated behaviours of Peyer's patch formation in the gut of a mouse embryo. PPSim 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.

2. Notations and metamodels

UML has an established abstract syntax that links behavioural and class concepts[2]; some features that relate state and class diagrams are summarised in Fig. 1. 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 express data structure, in terms of attributes and operations, that underpins OO models and implementation. Using a common metamodel allows us to confidently relate concepts between diagrams, and establishes the basis for model transformation (Czarnecki & Helsen 2006). Other key concept correspondences that are expressed in the full language definition 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.
– The guard or condition that must be true for a transition to occur is a Boolean clause.

[1] www.kennedy.ox.ac.uk/technologies/resources/ppsim-peyers-patch-development-simulator created by Kieran Alden.
[2] OMG's UML metamodels: https://www.omg.org/spec/UML/About-UML/

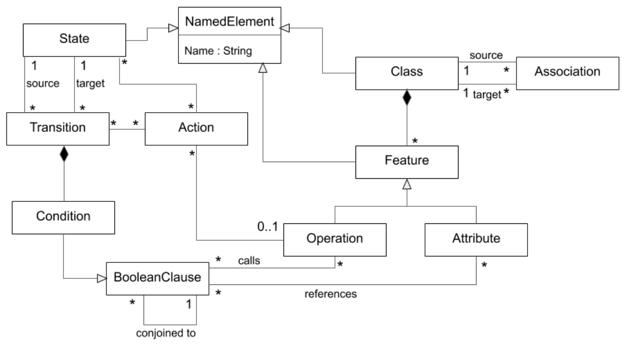


Figure 1 Part of the metamodel for state diagrams and class models in UML, based on UML 2.x and MOF metamodelling. The BooleanClause class is shown as referencing the Attribute class and calling the Operation class: in practice, Booleans reference the name or value of an object slot, and call the operations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

- 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 (by the need to evaluate the clause) execution of operations of the class or any class which is the target of an association from the class.
- An action on a state implies calls to class operations.
- A transition is triggered when the condition on the transition becomes true; as well as conditions external to the object, a transition condition may reflect that actions have caused the object's slot values to become inconsistent with its current state.

In designing an agent simulation, state diagrams are used to model agent behaviours, whereas, strictly, a class diagram would model agent types, operations and interactions. In general, agents are not synonymous with objects, but in practice, and because the developments target OO agent platforms, 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 in the case study state diagrams (Figures 2 and 3) represents object creation by an arrow (transition) from a solid black dot to a named state, represented as a soft rectangle. A state contains the state name and may list labelled actions that take place when an object enters or exits the state, or occur whilst the object is in the state; these actions can change the value of attributes of the object, reference linked objects, etc. The case study does not use the full UML transition syntax: a transition is an arrow from the source state to the target state, labelled with the condition on the transition.

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, simulation development seeks to faithfully replicate the uncertainty of an incompletely understood reality. Simulation validity is not binary, but is an argument that acknowledges assumptions and uncertainties – an argument that must be revisited if the domain understanding, the design, or the simulation purpose is modified.

The process of validation includes conventional software testing, and also trial-and-error tuning of the simulation so that the desired high-level behaviours emerge, without being explicitly programmed in, from behaviours and parameterisation that are an acceptable match to the real system. In the PPSim project, significant effort went into modelling the relevant parts of the biological domain and reviewing the models and fitness for purpose arguments with domain experts (fitness for purpose arguments are discussed in, e.g. (Ghetiu et al. 2009; Alden et al. 2011; Polack 2015; Stepney & Polack 2018)), to confirm their appropriateness to specific experiments. Compared to conventional software engineering, the published fitness for purpose arguments (Alden et al. 2011; Alden 2012) 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.

A validation step on design models, which is essential for MDE, is to check that the models conform to their metamodel. The case study state diagrams can be shown to conform to the metamodel in Fig. 2.1, and to be logically consistent (logical consistency rules form part of the full metamodel definition). For example, the literals or values in the Boolean clause representing a transition condition must be consistent with the definition of the source state of the transition; and, the conjunction of the Boolean clauses representing conditions on transitions from a state must be logically complete.

PPSim has demonstrated over the years that manual conformance checking is not sufficient. It is easy to create a conformant diagram that is not a valid representation of the domain. 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 (below) that describe the behaviour of PPSim cells. The fitness for purpose assurance needs to be extended beyond the design phase, to establish explicit mappings (traceability) between designs models and code. Model transformation could support demonstrable fitness by:

- reducing the risk of coding errors – any errors are in the transformation rules, and are thus systematic (and perhaps therefore easier to spot);
- providing repeatable coding — the same rules applied to the same diagram produce the same code; by extension, the same rules applied to an amended diagram would produce appropriately amended code;
- enabling quality simulation creation without advanced software engineering skills — the use of transformation (once the transformations have been written) frees the expert software engineer to focus on challenges such as finding representations that optimise computational efficiency with

understandability (of the models) and usability (of the simulator and its results).

4. The PPSim Design: state diagram models of cells

The original PPSim design (Alden 2012) was created in collaboration with domain experts led by Henrique Viega-Fernandes (see e.g. (Veiga-Fernandes et al. 2007)). The diagrammatic design comprises state diagrams for each of three implicated cell types (there is also an activity diagram, but it did not offer any additional insights to our work here)[3]. The simulation is a time-stepped execution of 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)).

In PPSim, the three cell types are known as LT_o, LT_i, $LT_{i(n)}$. There is a text discussion of the representation of each concept from the domain, and a comprehensive argument that the model is an appropriate model of the domain, given the hypotheses to be considered (Alden 2012). In this paper, in order to present a coherent narrative, the labelling of the state diagrams has been slightly simplified, but the biological labelling is retained, not least to emphasise that these are actual design diagrams, not academic examples.

In the simulator, the three types of cell are all located in a continuous space representing the mouse gut. In simple terms, the LT_o cells stick to the gut wall, whilst LT_i and $LT_{i(n)}$ cells move within the dominant direction of flow through the gut. LT_o and LT_i cells can bind a "RET Ligand". A bound LT_o produces "chemokine" which results in chemokine gradients in the environment; LT_i cells can detect the local chemokine and may move towards higher chemokine concentrations (chemotaxis). The notes accompanying the original state diagrams (Alden 2012) explain motion, contact and binding, the details of which do not concern us here.

4.1. LT_o Cells

Peyer's patch formation takes place around an active LT_o cell — one which can "bind a RET Ligand" — and requires formation of stable connections to cells around it. In the simulation, a LT_o cell is created *in situ* and does not move; other cells may come in to contact with or bind to a 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 on the former state. An active LT_o can contact of bind many other cells, and contacts are monitored by the cell: there are various thresholds at which changes of state are identified. The first state change occurs when the LT_o binds to a $LT_{i(n)}$ cell for the first time — the transition Touching LTin AND binding changes the cell state to Bound 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; in this state the LT_o can react to environmental chemokine ("chemokine upregulation"). After further non-state-changing contacts, the final transition occurs when a sufficient strength of adhesion (binding) and chemokine reaction is reached (though cumulated cell contacts); the LT_o is described as "mature" (Alden 2012) – in simple terms, the cell is now in a permanent cell cluster.

[3] Anonymous reviewers wondered why the transformation source is high-level state diagrams, and why we had not developed a class diagram in parallel: the simple answer is that the diagrams are the actual design models developed, some eight years earlier, and are typical of design models created in related projects. The diagrams have to be understandable to immunologists, in order to allow crucial review and checking (validation). There is no original class diagram: immunology comprises cells not objects, and the key feature of a cell is its behaviour; even if a class diagram had been created, it would not have been meaningful to the scientists, and could not have been validated.

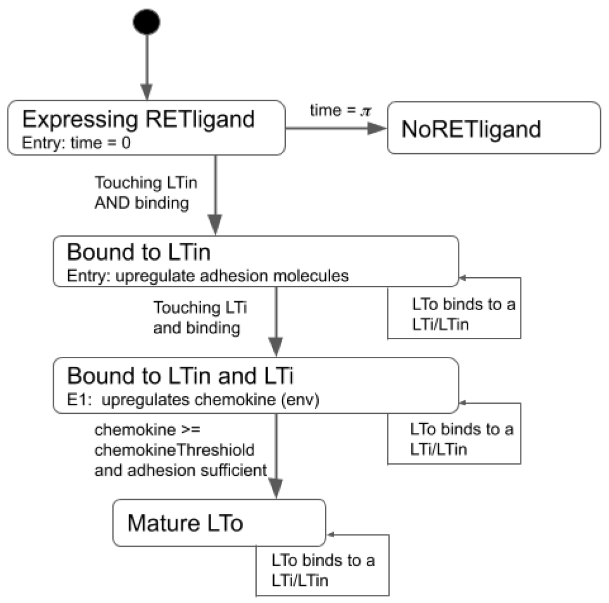


Figure 2 State diagram of LT_o cell, based on (Alden 2012). Notations and meaning: see Section 2.1.

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

LT_i and $LT_{i(n)}$ cells (Fig. 3) are mobile, and share many characteristics (at least in simulation), differing only in that a $LT_{i(n)}$ cell is not responsive to chemokine. Cell movement is modelled as an action during a state — the form of movement is different in each state, mediated by chemokine reception, when the cell is in contact with a LT_i, or 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. Some of the features provided are as follows.

- The simulation has a standard time step, and every agent is visited and updated in each step.
- There is an underlying spatial model, and agents have a location in the space.
- Experimentation may require measurement, encoded as data outputs.

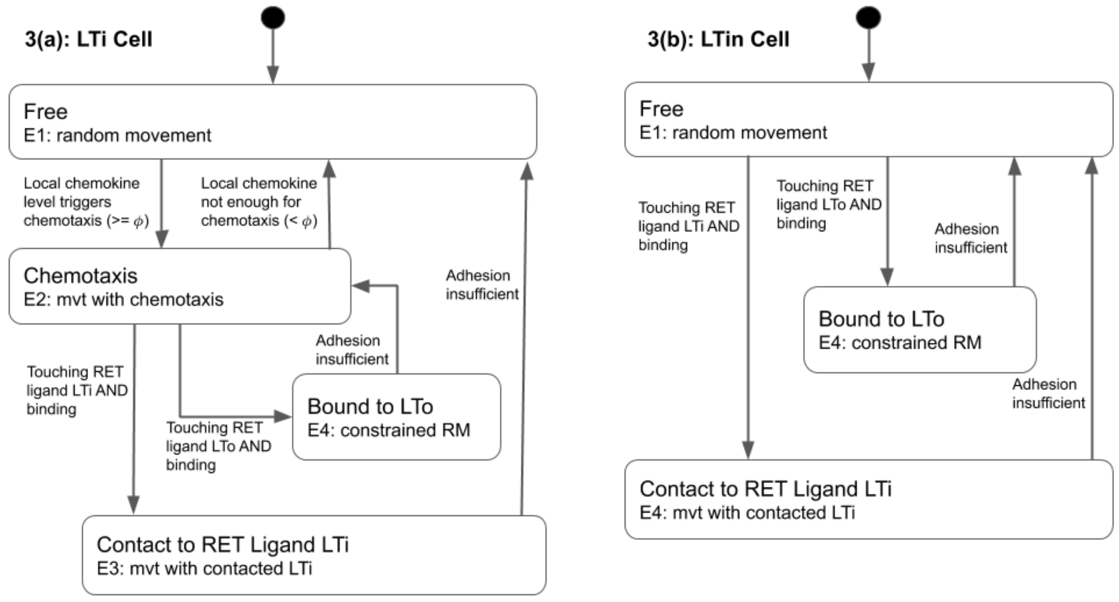

3(a): LTi Cell
Free
E1: random movement
Local chemokine level triggers chemotaxis (>= ϕ)
Local chemokine not enough for chemotaxis (< ϕ)
Chemotaxis
E2: mvt with chemotaxis
Adhesion insufficient
Adhesion insufficient
Touching RET ligand LTi AND binding
Bound to LTo
E4: constrained RM
Touching RET ligand LTo AND binding
Contact to RET Ligand LTi
E3: mvt with contacted LTi
3(b): LTin Cell
Free
E1: random movement
Touching RET ligand LTi AND binding
Touching RET ligand LTo AND binding
Adhesion insufficient
Bound to LTo
E4: constrained RM
Adhesion insufficient
Contact to RET Ligand LTi
E4: mvt with contacted LTi

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.

– 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

The rules proposed for a manual transformation from the published state diagrams to a class model exploit the metamodel relationships between concepts, Fig. 1. We propose five steps.

1. Use meta-information (which diagrams exist) to identify classes and potential 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 literals in Boolean clauses (conditions) and information on 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.

Manually-derived class diagram features are now summarised. Explanation for Steps 4 and 5 is in boxed text. Note that any step might identify features other than its main focus. Also, a derivation sometimes uses the explanatory text from (Alden 2012), in addition to the state diagrams.

5.1. 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
– LTi(n) class

However, it is easy to see that all three state diagrams relate to types of cell that share attributes and behaviours related to contact and binding, whilst mobile cells share movement characteristics. We can therefore deduce generalisations: Cell for the common features of all cells and MobileCell for the common features of LT_i and $LT_{i(n)}$.

5.2. 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 any time, at most one of an object's state attribute can be true; and (b) a partial order is needed to define 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 readable — an important consideration when models need to be 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).

5.3. Step 3: Identify associations implied by conditions

A UML state diagram captures the lifecycle of objects of one class but its conditions can reference properties of other objects

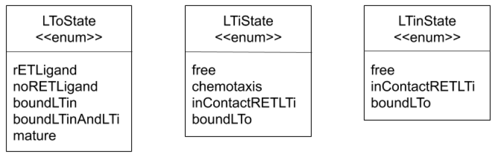


Figure 4 «enum» stereotypes defining the permitted states of cells shown in Figs 2 and 3.

or classes. Thus, 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, which can be modelled as an association to the parent class, MobileCell. Multiplicity is 1:m — one LT_o may bind any number of mobile cells; each mobile cell can bind to at most one LT_o.

The documentation accompanying the state diagrams indicates 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 map to optional 1:1 associations (an alternative would be to model these as attributes).

Turning to the mobile cells, both the LT_i and $LT_{i(n)}$ state diagrams have transition conditions relating to contact (Fig. 3). These conditions confirm the contact relationships identified for LT_o cells. The documentation also describes the various forms of movement of these cells, from which we can deduce that the location, and thus the identity, of a bound LTo is important.

5.4. Step 4: Identify class attributes

In addition to the state attributes (Step 2), attributes can be deduced from the state diagram conditions; the documentation accompanying the state diagrams helps to determine some of the details. To avoid making arbitrary decisions, the types of attributes are descriptive (e.g. adhesion-related attributes have a type, AdhesionType), unless a type such as Integer or Boolean is obvious. Systematic analysis of conditions results in four attributes, as follows.

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

5.5. Step 5: Deriving class operations

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 operations 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.

Step 4 Details:
The relevant conditions from the state diagrams are:

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

time is a local timer on a LT_o cell which can be incremented to a threshold, π. The documentation states that π is a simulation-level parameter.
The documentation also describes the derivation of chemokine gradients and associated initial and maximum values, using simulation parameters whose values are determined by calibration. A LT_o cell's chemoExpressionLevel is calculated from a chemokine curve, whilst a LT_i cell calculates the localChemokineLevel based on the position and chemoExpressionLevel of its nearest LT_o.
The documentation explains that cells have a probability of adhesion that increases with duration of contact; all cells have an adhesionExpressionLevel that is initially set to 0. There are simulation parameters to establish how the adhesion level is set and incremented.

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()

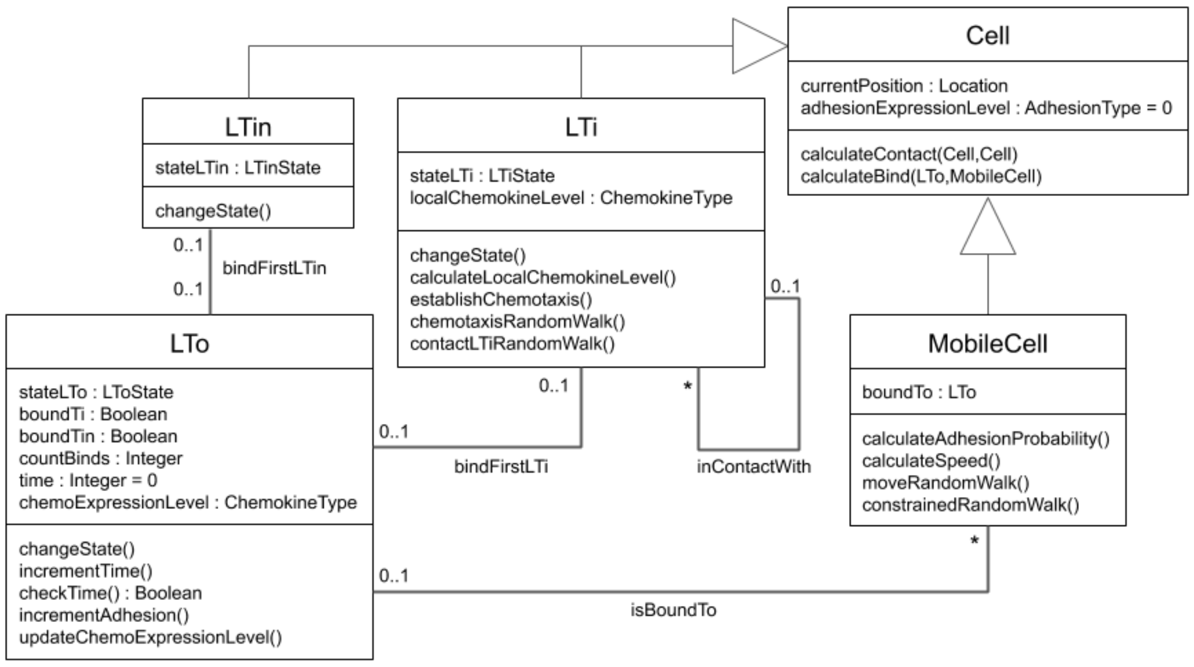


Figure 5 Derived class diagram (omitting setters and getters).

– LTi.establishChemotaxis()

These operations also support many of the actions on the states in the state diagrams.

Step 5 LT_o Action Details:
The LT_o action Entry: time = 0 has been accounted for by setting the initial value of the time attribute to 0.
The LT_o action, Entry: upregulate adhesion molecules corresponds to the operation to incrementAdhesion().
The LT_o action, E1: upregulates chemokine (env), is captured by updateChemoExpressionLevel().

Operations to implement motion are derived from actions on LT_i and $LT_{i(n)}$ cell states:

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

The last derived operation, LTi.contactLTiRandomWalk() identifies a missing association: if the movement of a LT_i cell can be influenced by another LT_I cell in which it is in contact (Fig. 3), then there must be an association recording which other LT_i cells a specific cell is in contact with.

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)

Step 5 Motion Details:
From the documentation accompanying the state diagrams (Alden 2012), the speed of a mobile cell is drawn at random from a Gaussian distribution, and determines how far the cell moves in each time-step. The form of motion is determined as follows:

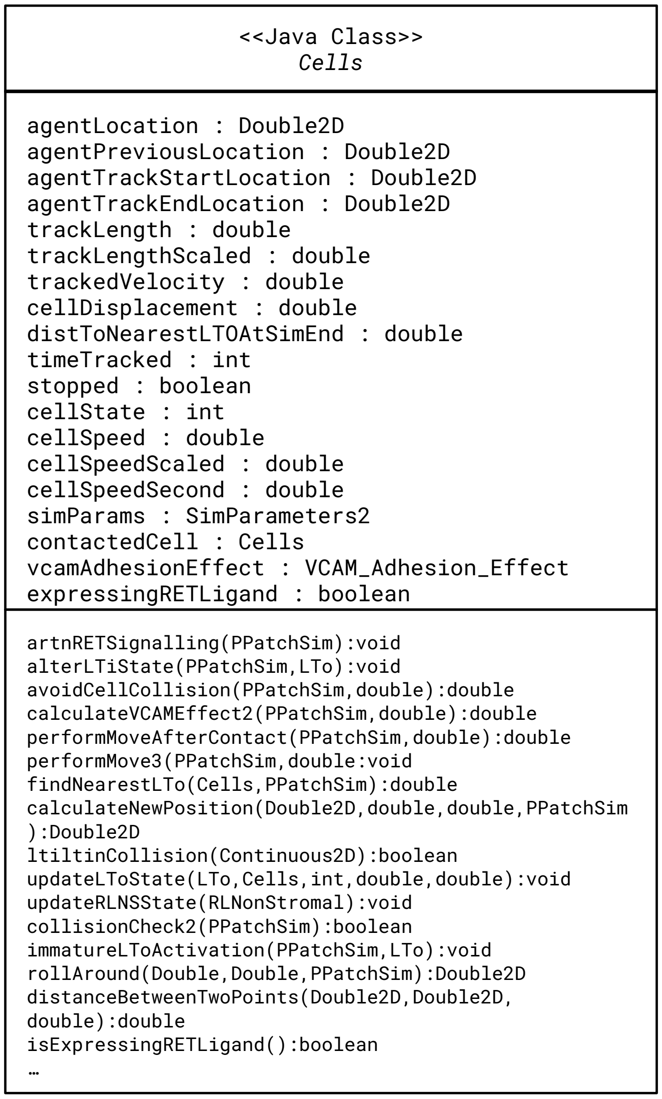
– as a random walk (E1: random movement in Fig. 3), until cell interactions start;
– as a chemotaxis-weighted random walk (E2: mvt with chemotaxis) in which the probability of moving in any direction is related to chemokine strength, calculated from the relative location of a LT_o emitting chemokine;
– as a random walk weighted by the adhesion strength and a probability that a bound cell can move away from the binding LT_o (E4: constrained RM). The movement of a LT_i may also be constrained according to action E3: mvt with contacted LTi, where two LT_i cells are in contact. This form of movement, like the chemotaxis-weighted random walk, only applies to LT_i cells.

Note that 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 any form of sequence diagram that describes composite operation details.

The full set of derived classes, attributes and operations is shown in Fig. 5, above.

6. Validating the derived model

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



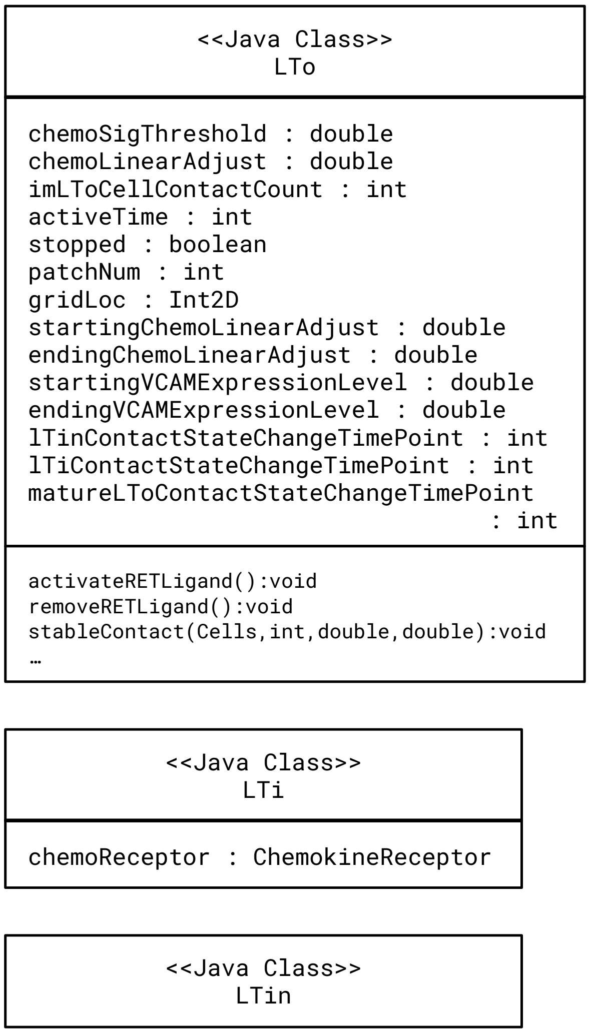
```
<<Java Class>>
Cells

agentLocation : Double2D
agentPreviousLocation : Double2D
agentTrackStartLocation : Double2D
agentTrackEndLocation : Double2D
trackLength : double
trackLengthScaled : double
trackedVelocity : double
cellDisplacement : double
distToNearestLTOAtSimEnd : double
timeTracked : int
stopped : boolean
cellState : int
cellSpeed : double
cellSpeedScaled : double
cellSpeedSecond : double
simParams : SimParameters2
contactedCell : Cells
vcamAdhesionEffect : VCAM_Adhesion_Effect
expressingRETLigand : boolean

artnRETSignalling(PPatchSim):void
alterLTiState(PPatchSim,LTo):void
avoidCellCollision(PPatchSim,double):double
calculateVCAMEffect2(PPatchSim,double):double
performMoveAfterContact(PPatchSim,double):double
performMove3(PPatchSim,double:void
findNearestLTo(Cells,PPatchSim):double
calculateNewPosition(Double2D,double,double,PPatchSim
):Double2D
ltiltinCollision(Continuous2D):boolean
updateLToState(LTo,Cells,int,double,double):void
updateRLNSState(RLNonStromal):void
collisionCheck2(PPatchSim):boolean
immatureLToActivation(PPatchSim,LTo):void
rollAround(Double,Double,PPatchSim):Double2D
distanceBetweenTwoPoints(Double2D,Double2D,
double):double
isExpressingRETLigand():boolean
…
```

Figure 6 Cells superclass generated from PPSim code (omitting setters, getters and features needed to run Java Mason)

Since the Java Mason PPSim code exists, we can apply a more independent validation check, by comparing the class diagram derived manually (referred to as the *derived model*) and the class structure of the existing PPSim code (the *code model*), which can be extracted by any development environment capable of representing code structure as a UML-style class model[4]. The code model extracted from the PPSim code includes setters and getters, making the image too large to reproduce clearly. The key agent features are reproduced in Figs 6 and 7.



```
<<Java Class>>
LTo

chemoSigThreshold : double
chemoLinearAdjust : double
imLToCellContactCount : int
activeTime : int
stopped : boolean
patchNum : int
gridLoc : Int2D
startingChemoLinearAdjust : double
endingChemoLinearAdjust : double
startingVCAMExpressionLevel : double
endingVCAMExpressionLevel : double
lTinContactStateChangeTimePoint : int
lTiContactStateChangeTimePoint : int
matureLToContactStateChangeTimePoint
                                : int

activateRETLigand():void
removeRETLigand():void
stableContact(Cells,int,double,double):void
…
```

```
<<Java Class>>
LTi

chemoReceptor : ChemokineReceptor
```

```
<<Java Class>>
LTin
```

Figure 7 Cell Classes generated from PPSim code (omitting setters, getters and features needed to run Java Mason)

6.1. Comparing the derived model and the code model

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 and some have wider scope than in the derived model. The different placement complicates class-by-class comparison between the derived and code models. Instead, the following addresses the generic concepts in turn: state and time, chemokines

[4] 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. Because the case study is an existing laboratory simulator, this has been extensively studied as part of the original development: PPSim was subject to extensive calibration and sensitivity analysis (Alden 2012; Alden et al. 2012), using Alden's open-source simulation analysis tools (Alden et al. 2016).

and adhesion, movement, contact and binding. Again, derivation details are presented in boxed text.

6.2. Derived and code models of state and time

The code model has a generic integer state variable `Cells.cellState`, whereas the derived model has a state attribute, with an enumerated type, on each cell subclass. Whilst neither model is inherently wrong, the derived model allows traceability to the design and the biological domain, whilst the specific state change operation on each cell class facilitates calling the class operations to execute checks and run actions. The inherently ordered integer type used in the code model helps in coding state change, but the code model does not cleanly map to the state changes in the state diagrams — there are many operations encoding behaviour that causes or arises from state changes, which seem to be coding aspects of biological descriptions, rather than the design in the state diagrams.

Both derived and code models have an attribute on LT_o expressing its local timer. However, where the derived model provides distinct operations for the time attribute (LTo.incrementTime() and LTo.checkTime()), checking time in the code model is conflated with the effect of exceeding the time limit.

Details of cell state change, code model:
The code model Cells operations that change the state are:

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

Details of time, code model:
Code model operations that reference `LTo.activeTime` are `activateRETLigand()` and `removeRETLigand()`: the checking of time is conflated with state changes.
The code model also has three timing attributes to record contact duration, which are used in calculating bindings (e.g. by the operation `LTo.stableContact()`). This is below the level of detail shown in state diagrams.

- `LTo.lTinContactStateChangeTimePoint`,
- `LTo.lTiContactStateChangeTimePoint`,
- `LTo.matureLToStateChangeTimePoint`.

The code model also includes `Cells.timeTracked:int` and `LTi.getTimeTracked()`, which support experimental data collection.

6.3. Derived and code model chemokine representations

The code model again encodes details of chemokine handling which are not taken from the design diagrams, whereas the derived model proposes ChemokineType as a placeholder for an abstract data type which, conventionally, would capture the basic chemokine operations as well as the attribute type.

The derived model provides clear traceability and separation of concerns, though we do not explore here whether this conceptual clarity would complicate the encoding of low-level behaviours in the OO agent platform. In software engineering terms, the use of an abstract data type would generally be considered to be a more appropriate solution than a direct encoding of type-related behaviours.

Details of chemokine handling, code model:
The code model `LTo` captures chemokine levels at both cell and system level, in attributes `chemoSigThreshold`, `chemoLinearAdjust`, `startingChemoLinearAdjust` and `endingChemoLinearAdjust`.
The evaluation of local chemokine by `LTi` cells is captured in the movement operations, which are on the `Cells` superclass in the code model.

6.4. Derived and code models of contact and binding

Binding is fundamental to Peyer's patch formation. PPSim uses the concept of adhesion to encode binding strength and, since it is common to all cells, both the derived and code models place adhesion-related features in the Cell superclass. In the derived model, AdhesionType is again a place holder for an abstract data type, whereas the code model again provides operations that express detail from the documentation.

Details of adhesion:
In the derived model, MobileCell.calculateAdhesion-Probability() calculates the adhesion probability factor. Operation Cell.calculateBind() compares this to the Cell.adhesionExpressionLevel to determine whether adhesion is sufficient to maintain a bind. Cell.adhesionExpressionLevel is updated by LTo.incrementAdhesion().
In the code model, the encoding captures the biochemistry of binding, referring to "VCAM expression": attributes `LTo.startingVCAMExpressionLevel`, `LTo.ending-VCAMExpressionLevel`, `Cells.vcamAdhesionEffect`, and operation `Cells.calculateVCAMEffect()`. The type `VCAMAdhesionEffect` suggests that the code uses an abstract data type for adhesion, but the detail is lower-level than in the derived model.

The approaches to recording contact and binding, and the associated calculation structures, follow different design approaches but support a similar computation. In the derived model, the Cell superclass has the operations to calculate contact and binding (calculateContact(), calculateBind()), Whilst the code model provides `Cells.contactedCell` to identify the cells that are in contact with a particular cell.

Details of contact and binding:
The derived model records that a LT_o has the achieved required bindings using attributes, `LTo.boundLTi:Boolean` and `LTo.boundLTin:Boolean`, providing operations to check bindings; associations allow interrogation of the link to find which cell(s) are bound.
The code model has a generic operation which allows any cell to find its nearest LT_o cell, `Cells.findNearestLTo()`. The approach avoids the need to record that a cell is in contact, and subsequently to determine which LT_o cell is the current focus of movement. In the code model, the encoding of contact follows the documentation (Alden 2012) — cell centres must be within the sum of half their respective diameters; the code model also captures the documented calculation of binding strength for any cells in contact with a LT_o, using adhesion level and a generated random number (to maintain stochasticity of binding).
The code model attribute, `LTo.imLToCellContactCount` plays the same role as LTo.countBinds in the derived model. The *im* prefix is a reference to the biological term, *immature*, which describes 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 transitions.

6.5. Derived and code models of movement

The basics of movement (moving a mobile cell to a new location, according to its speed and the constraints imposed by binding) are similar in both models. In the derived model, we follow the state diagrams in assigning movement only to MobileCell classes, providing operations for each form of cell movement. These operations need to access cell locations, binding, and adhesion data. By contrast, the code model again encodes lower-level calculations described in documentation; all the movement behaviours are on the superclass, `Cells`, even though LT_o cells do not move.

7. Discussion

The first point to be made in discussing the manual derivation of the class model is that, whilst the derivation is an academic exercise, both the state diagrams that are the starting point for derivation, and the Java Mason code that is the target are real. The fact that the source and target are part of a documented project with published research results. It is easy to assert that a different starting point would have produced a cleaner derivation, but our goal here is to explore transformation from existing diagrams. One issue that this highlights is just how many inconsistencies arise when using software engineering techniques such as UML modelling without tool support.

The derivation shows that, in principle, state diagrams (even quite abstract ones) can be mapped into at least a partial class diagram, with the structure and features needed to express cell agents and behaviours. However, the validation of the derived class model shows that the derived class model is not sufficient to support transformation to code, and does not provide the agent-implementation information that would be needed to derive a Java Mason implementation. This is unsurprising, since we made no attempt to model the Java Mason "agent language", or to analyse how the platform encodes the agent concepts, behaviour, parameterisation, visualisation and data collection.

Details of movement: code model:
In the code mode, the key movement-related attribute is `CellSpeed`, but there are additional speed attributes, `Cells.cellSpeedScaled`, `Cells.cellSpeedSecond` and `Cells.trackedVelocity` for use in experiments.
The code operations are at a lower level than those in the derived model, and focus on the effect of movement (to relocate the cell) rather than the movement itself:
`performMoveAfterContact()`,
`performMove()`,
`calculateNewPosition()`,
`rollAround()` and
`distanceBetweenTwoPoints()`. The code model's `performMove()`, `performMoveAfterContact()` and `rollAround()` implement the unconstrained 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 for detecting and avoiding cell collisions (`Cells.lTiLTinCellCollision()`, `Cells.collisionCheck()` and `Cells.avoidCellCollision()`), behaviour that is not expressed in the state diagrams.

Taking a different perspective on the insufficiency of the derived class model, we can envisage potential next steps to include developing a metamodel for Java Mason, and exploring transformation workflows with intermediate models. However, we can also draw insight into the engineering ideal for behavioural design models. The PPSim state diagrams do not provide sufficient detail for code generation, whilst the accompanying documentation does not have the formal structure needed to support model-management. To develop MDE support for simulation development — at least in the context of the wider CoSMoS process with its emphasis on demonstrable fitness for purpose — it would be useful to record some more explicit conventions and guidance on the software engineering detail that software engineers need to provide in behavioural design models. CoSMoS has focused on assuring the expression of domain information in the design; automated coding would need to additionally assure the inclusion of sufficient computational detail in design.

The attempt to validate the manually derived class model highlights the creativity of manual coding. In PPSim, the programmer has created code that maps to the domain model, but does not always follow the design expressed in the state diagrams. Use of MDE model transformation would force a fundamental change, potentially facilitating traceability and demonstration of the fitness for purpose at the cost of removing most opportunities for creative coding. However, it is also likely that transformation could improve code quality and facilitate creation, maintenance and reuse of code. A side-effect of removing creativity from coding is to enable creative modelling and transformation solutions that are reusable across simulation designs.

Looking in more detail at the steps of the manual derivation, the systematic approach relies on meta-information such as the *existence* of state diagrams, and on intuition (in generalisation). However, it is arguable that, for code generation, the class structure could be derived without reference to meta-information and that generalisation can be retro-fitted by post-hoc review of features that classes have in common (as is often done in manual coding).

A challenge that the derivation faces, but the native coder did not, concerns associations. It is straightforward to identify the need for an association, but there is no direct way to encode associations: object linkage cannot be captured in code at class level. We might conclude, therefore, that there is no point in deriving associations. Indeed, observing that the hand-crafted code uses the (unstructured) documentation to create many of the computational details, rather than the (well-defined) state diagrams, we would recommend that effort should focus on systematically identifying *message passing* to support the calling of derived operations. Indeed, it seems that sequence diagrams, or similar, that express the message flows that enact behaviour might be necessary for transformation. This should not be surprising, since software engineering has long recognised that early effort in modelling and validation facilitates quality software development outcomes. Tying down the behaviour specification early in development leaves fewer open options in coding, and makes design decisions easier to record and analyse.

The derivation of classes exploits all the abstract concepts of the state diagrams, and applies the conceptual equivalences defined in the metamodel. However, the manual derivation also needs an intuitive understanding of how conditions and actions reference object features. The manual approach is relatively easy to describe, but, because a condition may 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 documentation accompanying the original state diagrams was repeatedly used to complement the 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.

Looking in more detail at the code model, there are clear reasons why the code includes detail that is not, and cannot be, captured in the design models. The code-model classes include attributes and operations used to: (a) record attributes and operations needed for experimental results; (b) operate the simulator (stop and start runs); and (c) support different experimental set-ups of the simulator. An automated transformation assumes that only the design models are required to derive code, so further work is 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.

In attempting to validate the derived model against the code model, the effect of open design decisions is very evident. It is arguable that code derived from the diagrams would be better structured than that manually coded — it would certainly have better traceability and maintainability. Transformation would also remove some of the terminology issues, where the code-model naming differs from the state-diagram naming conventions. More specifically, transformation would eliminate the possibility of a concept having different names in design models and code.

It is interesting to note that the detailed documentation accompanying the design and the PPSim code model (Alden 2012) 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 at implementation: fitness for purpose argued at the domain and platform model levels would pertain by transformation to the code.

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 design problems rather than just creating code. We have confirmed by manual transformation that the design contains a part of the information needed to create code. Ideally we would like to have more algorithmic behavioural models (sequence diagrams) that formalise the detail of behaviours captured in the documentation, making it accessible to transformation. If the designers know that this is desirable, effort could be put into converting documented calculations into transformable diagrams. There are, however, still aspects where 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, although the manual transformation made different design decisions and used different approaches to supporting the same behaviours. We speculate that 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 be focused 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.

A caveat on the current work on model transformation support is that, with the increasing recognition of agent simulation as a tool for research on complex systems, the widely-used OO (Java-based) 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, and this may lead to adjustment of parameter values, or even adjustments to the code. 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 parameter values and behaviours. Better engineering support for 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

Our work on simulation engineering is part of a long-running effort involving many people. In particular, we would like to acknowledge the contributions of Susan Stepney and Jon Timmis and the CoSMoS project team (EP/E049419/1). We would like to thank the two anonymous reviewers of this paper, for their very insightful comments, and their acceptance of our detailed description of how not to go about designing a model transformation. We hope that Martin Gogolla will appreciate our attempts at MDE on behavioural models, and our discussion of validation, both of behavioural models and of our final product.

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)

Greaves, R. B., Read, M., Timmis, J., Andrews, P. S., Butler, J. A., Gerckens, B., & Kumar, V. (2013). In silico investigation of novel biological pathways: the role of CD200 in regulation of T cell priming in experimental autoimmune encephalomyelitis. *Biosystems*, *112*(2), 107–121. doi: 10.1016/j.biosystems.2013.03.007

Moore, J. W. J., Moyo, D., Beattie, L., Andrews, P. S., Timmis, J., & Kaye, P. M. (2013). Functional complexity of the Leishmania granuloma and the potential of in silico modelling. *Frontiers in Immunology*, *4*(35). doi: 10.3389/fimmu.2013.00035

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., & 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

Williams, R. A., Greaves, R., Read, M., Timmis, J., Andrews, P. S., & Kumar, V. (2013). In silico investigation into dendritic cell regulation of CD8Treg mediated killing of Th1 cells in murine experimental autoimmune encephalomyelitis. *BMC Bioinformatics*, *14*, S6–S9. doi: 10.1186%2F1471-2105-14-S6-S9

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. She works on software engineering of demonstrably fit-for-purpose simulations of complex systems, building on the work of the CoSMoS Project. 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-source 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/.