Reflecting on Open-Ended Evolution

Susan Stepney and Tim Hoverd

YCCSA, University of York, YO10 5DD, UK
susan@cs.york.ac.uk

Abstract

We describe a computationally reflective object-oriented architecture suitable for incorporating open-ended innovation and emergent entities into simulations. This allows emergent properties to be reified into objects. This requires modifying the model, and the metamodel, by incorporating novel classes and metaclasses dynamically. The classes and metaclasses are modified by including them in the model through reflection. We argue that such computationally reflective introduction of novelty is necessary for true open-ended simulations.

Introduction

Open-ended dynamics, supporting constant novelty generation, is a goal of ALife simulation.

Open-ended evolution has been defined as "a process in which there is the possibility for an indefinite increase in complexity" ([20], which also contains a comprehensive review of the concept in biology). Bedau [2] talks in terms of systems that exhibit "supple adaptation", which involves them "responding appropriately in an indefinite variety of ways to an unpredictable variety of contingencies". Open-ended novelty generation and evolution are features of biological life, but are proving hard to achieve in silico.

Classical evolutionary algorithms, with their fixed genome representations, can produce new things only within that limited representation. Evo-devo algorithms break out of this limitation, by allowing a genome to develop into a phenotype, but they are still confined to a single (albeit much richer) representation.

The desired continual increase in complexity is not merely a constant supply of new things (variations of a theme), or even of new kinds of things (speciation), but of new kinds of new kinds of things (major transitions, radical novelty, novel concepts). In computational terms, we might say we need a constant supply of new objects (the new things), new classes (new kinds of things, new representations), and new metaclasses (new kinds of kinds of things, new kinds of representations).

Here we take a computational modelling view of the problem, and describe what we believe are minimal requirements for true open-ended dynamics in simulations: simulations that can modify their own model and metamodel as they execute. This implies that they can modify how they modify themselves. One key step on this route is the need to reify ("make concrete, or real") emergent properties, as these are a rich source of novel concepts outside the language of the pre-existing system.

The structure of the rest of the paper is as follows. First we discuss the process of reifying emergent properties, both at the class and metaclass levels. Then we describe how a computational system can modify its own model and metamodel at runtime. Finally we specify a bootstrap architecture for such a self-modifying system.

Extension, Intension, and Emergence

Consider an agent-based flocking simulation, implemented in some object-oriented (OO) programming language. A collection of boid objects exhibits various behaviours, and potentially forms flocks.

Assume the individual boid objects have names, eg Tweety, Cheeky, Polly, and ages, eg juvenile, adult, old. We can define particular sets of boids in two ways. An *extensional* definition explicitly enumerates the members: $A = \{\text{Tweety}, \text{Polly}\}$. An *intensional* definition is an implicit definition of membership in terms of properties of the members: $B = \{\ b : \mathsf{Boid} \mid b \text{ is juvenile}\ \}$.

In an atemporal world of pure logic, a property is eternally either true or false, so extensionally defined set A and intensionally defined set B are either equal or not equal (have precisely the same members, or do not), and the difference in definitional approach is logically unimportant[1]. However, when properties are a function of time (as with stateful objects), an intensionally defined membership need not be static (for example, the membership of B may change as boids age). Hence A may equal B at one time, but not at another. In such a case, we need to be clear about whether the

[1] Except for such paradoxical definitions as "the set of all sets that are not members of themselves", and other issues underpinning the foundations of mathematics, but we are not addressing these issues.

extent or the intent is the relevant defining property of our set of interest (for example, are we interested in Tweety and Polly, and "juvenile" is just a convenient shorthand for denoting them at this moment; or are we interested in juveniles, who just happen to be Tweety and Polly at this moment).

In general, we are interested in intensional property-based definitions, in potentially-changing collections of things that have certain properties in common (such as "all the blue birds more than a year old"), rather than in explicit but arbitrary collections (such as $\{\text{Tweety}, (\emptyset, \text{Rover}), 42\}$). And we are more interested in generic intensionally-defined concepts ("flock"), than in specific one-off extensional collections ("those birds over there").

In an OO program, nevertheless, collection objects (instances of Dictionaries, Sets, Lists, etc) are almost always extensional: they are static collections of the actual objects. The intent of such sets is only implicit (not captured in the code, except maybe through invariants or contracts), and much coding effort goes into maintaining this intent (explicitly adding and removing objects from the otherwise-static collection). This intent-implementing code, with its property-checking component, can be encapsulated inside a class. For example, consider the set of "all instances of class X". This is an intensional definition: the set will contain different elements at different times, as instances of class X are created and destroyed. So in Smalltalk-80 [7] the (class) method allInstances returns an extensional set of all the instances of the class at the time of the message-send. The set itself does not change as objects are created and destroyed: a new message needs to be sent to the class to find the current value. The implementation hides the details of how this set is constructed each time; logically it is equivalent to constructing the set by examining every object and testing for the defining property.

Emergence as implicit intension

Now consider the OO boid simulation. We point to an area of the screen, and say, "the flock is those boids". So at any given moment, a flock appears to be an extensionally-defined set of boids: *flock* = {Tweety, Cheeky, ..., Polly}. However, unlike a true extensional definition, the membership of the flock set can and does change, as boids leave and enter. This demonstrates that the flock is 'really' intensionally defined: *flock* = $\{ b : \text{Boid} \mid b \text{ has property f} \}$. We just do not know what the intensional property f is, in advance[2]. The flocking property is emergent.

In some sense a flock is a 'thing', but it is not an object in our simulation, and there is no Flock class with which to capture and hide the intent-preserving code that tracks this set as boids enter and leave the flock. (Of course, we could have defined such a class, but that would require us to know beforehand the emergent properties; we assume here that we have not.)

We need some way to add this class and its intensional definition to the model and simulation *as and when the property emerges*. First we discuss different degrees of intensionalisation, and then a method and architecture for modifying the simulation with novel emergent properties.

Intensionalising Emergence

We reify a specific flock by capturing it as an extensional object in the simulation, for example, as an instance of some generic Collection class (*theFlock* = collectionInstance$(b_1, \ldots, b_n)$). We can define the concept of flock in a new class Flock that explicitly captures the emergent intensional property, and so intensionalise the flock: *theFlock* = flockInstance$(b_1, \ldots, b_n)$. We can intensionalise an emergent property in a simulation in the following three ways, yielding different dynamics in the resulting system.

External Instrumentation

Ordinary agents might remain blind to the existence of the emergent: it has no direct effect on them. For example, boids in a simple flocking simulation react to other boids independent of whether they are in a flock. (That is, their behavioural *rules* are unchanged, although of course their resulting *behaviour* is sensitive to the existence of the flock.)

In a simulation, we might add a FlockRecogniser subsystem, including a class FlockTag whose instances tag the detected flocks, and merely provide statistics on the simulation's behaviour. Such instances would have no effect on the individual boids' behaviour, whether within or outside a flock.

Internal Detection

External instrumentation is the least interesting kind of reification, as the emergent is explicitly visible only to external observers. Crutchfield [5] talks about "intrinsic emergence", where there are internal observer processes that can "take advantage of the emergent patterns".

The next level of reification includes internal detection, whereby ordinary agents notice the existence of the emergent, and change their behaviour based on it. For example, a more sophisticated flocking simulation could have boids modified to be able to sense and interact directly with flock objects, preferring to move closer to a flock than to boids not in a flock, say. The flock object exists in the simulation, but is merely a derived consequence of the boids' behaviours: it has no active behaviour of its own, it merely influences the behaviour of other objects.

[2] Additionally, the property is probably somewhat fuzzy. For example, consider what might be the minimum size of the set *flock*. One boid, even two boids, do not make a flock. It has no well-defined answer; a flock is a fuzzy concept. (See, for example, the description of the Sorites Paradox in [11].)

Reification

With full reification of the emergent, ordinary agents notice the existence of the emergent, change their behaviour, and are also directly affected by it. The emergent becomes an intensional entity in its own right. Being a component of the emergent then stops being defined merely as an extensional property (happening to being in the correct location to be in the extension, say), and becomes something that is granted by the emergent entity (membership rules, say).

For example, a reified flock object in a simulation might actively prevent boids from entering or leaving the flock: it would then be acting as a kind of 'membrane' around the flock. (We are *not* suggesting this happens in real-world flocks. Here we are simply exploring the kinds of ways that a simulation might react to the presence of an emergent: we are interested in getting complex open-ended dynamics in the simulation, not in faithfully replicating how such processes occur in the real world.) The reified emergent becomes available in the simulation to be a first class component in further (higher-level) emergent behaviours.

The effect of the reified emergent on its constituent members could be considered to be a form of downward causation [3, 22]. Although such a concept is anathema to physicists, it is an everyday notion to sociologists. Reification of some societal constructs changes membership properties (for example, citizenship) from extensional (happening to be located in the country) to intensional (having the conferred property of being a citizen) in exactly this way.

Intensionalising Emergence internally

We have discussed modifications to the simulation to achieve several kinds of intensionalisation, to capture emergent properties as explicit entities within the simulation. In this section we propose how to achieve this *dynamically within the simulation*, through the use of computational reflection [16].

Models

When writing a program, it is good software engineering practice to write a model of the program. For an OO program, that model is often written in an OO modelling language such as UML, identifying the classes, associations, interactions, behaviours, and so on. This model provides the abstract language of the concepts to be implemented in code. Even if no such model is written explicitly, it is implicit in the structure and dynamics of the written and executing code.

For example a (very simplified) class model of an agent-based boid simulation might look like figure 1. This is a model of the implemented code. Emergent (unimplemented) properties do not appear in this kind of model.

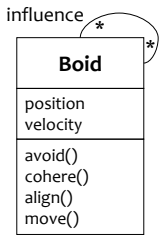

Figure 1: A (very simplified) UML class model of a boid simulation. There is a single Boid class, listing the attributes and operations of boids. Each boid has zero or more boids that influence it. (It bases its behaviour on the attributes of these boids, but that is not captured in this model.)

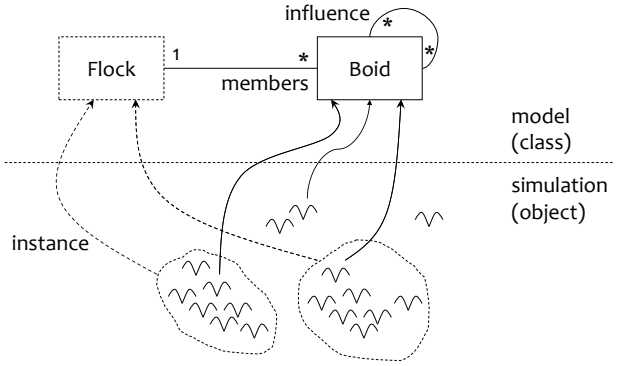

Figure 2: Model of Boids and emergent Flocks

Emergent classes

Although the model of the simulation code does not include emergent concepts, we can build a (different) model that does. In this new model, the emergent is captured as an extensional object; it can then be intentionalised (its defining property captured in a class definition).

So we augment our model with an emergent class (which we draw as a dashed class box)[3]. This class captures the emergent property, and its instances. Figure 2 shows two levels: a model level with a normal class Boid and an emergent class Flock. We also show an object level view (a snapshot of the objects present during execution). The boid objects are instances of the Boid class. Some boid objects are members of flocks. We say that these emergent flock objects are instances of the emergent class Flock.

The emergent class might be a subclass of an existing 'ordinary' class in the model. For example, in an evolutionary system, a new kind of mutation operator might emerge ([8] discusses an example of an emergent macromutation, figure 3). In such a case we assume that the superclass is *abstract*, with neither intensional nor extensional instances of its own. On the other hand, the emergent class might be

[3]This is not part of UML, and so is an extension of the modelling language.

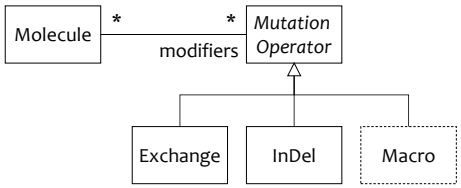

Figure 3: An example of an abstract superclass with ordinary and emergent subclasses (derived from [8], which has predefined Exchange and InDel mutation operators, and exhibits an emergent macromutation)

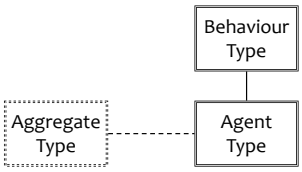

Figure 4: A (very simplified) metamodel of agent based models. There is an Agent Type (an instance of which is the Boid class), and a Behaviour Type (instances include the boids' avoid and align behaviours). This metamodel has been augmented by an emergent Aggregate Type (an instance is the emergent Flock class).

a genuinely new kind of concept in the model, with no pre-existing superclass.

Once we have augmented our model with emergent objects and classes, we could build a new simulation with them as coded classes. But for an open-ended simulation, we need a system that can itself recognise such entities, and change its own model, *at run-time*, to include such intensionalised emergent classes dynamically.

Metamodels

Changing the model (to allow for new kinds of executing objects), although necessary, is not sufficient for full open-endedness. We also need to change the metamodel, to allow new kinds of things in the model.

In an analogous way to how a model provides the language for writing the code, a metamodel provides the language for writing a *model*: it defines the kinds of things that can occur in the model (it is the model of the model). UML's metamodel includes concepts such as class and association. An agent-based modelling language metamodel would include concepts such as agent and behaviour. In the same way that models need to be augmented to include emergents, so do metamodels (figure 4).

Models and instances form a two-level modelling architecture. The Object Management Group (OMG) uses a four level modelling architecture [12, ch.8]: M0 = base instance (the objects in the simulation); M1 = model (defining the kinds of things in the simulation, such as Boid, Ant; written

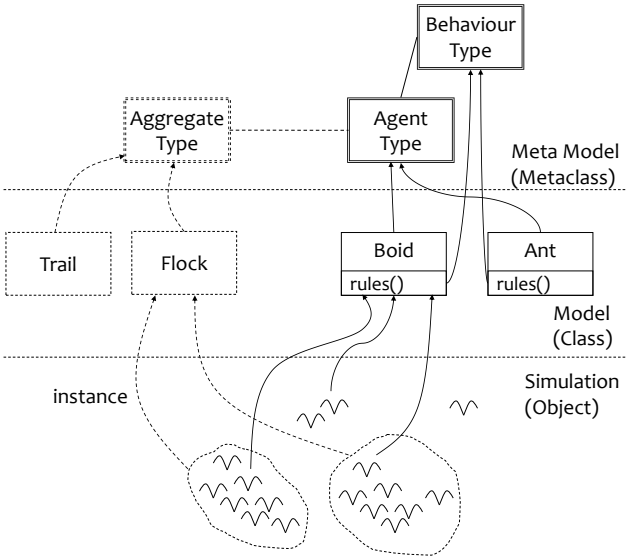

Figure 5: Metamodel and Model of an agent-based simulation

in, for example, UML); M2 = metamodel (defining an ontology, the kinds of thing in the model, eg Class and Association for UML models, AgentType for agent-based models (ABMs); also written in for example UML); and finally M3 = meta-metamodel (defining the kinds of thing in the metamodel, written in, for example, OMG's Meta Object Facility (MOF) language). Infinite regress is avoided by allowing the meta-metamodel to be written in MOF. Here we consider only the bottom three layers, M0-2.

Another example of this four level architecture is: M0 = executing program; M1 = a Python program; M2 = the Python programming language; M3 = BNF and denotational semantics. Changing the model is analogous to changing the *program*; changing the metamodel is analogous to changing the *programming language*.

Emergent Metamodels

The metamodel of an ABM (figure 5) describes the kinds of things in an agent-based simulation: it has a metaclass AgentType. The emergent class like Flock in the model also needs a metaclass: it is an emergent AggregateType. So there can be emergent metaclasses too (where an emergent class is not an instance of some existing metaclass).

Speciation and major transitions

We have seen three main kinds of reification:

1. Reifying an emergent subclass (for example, the macromutation class in figure 3). The concept already exists in the model (the superclass); the reified subclass is a variant of that concept.

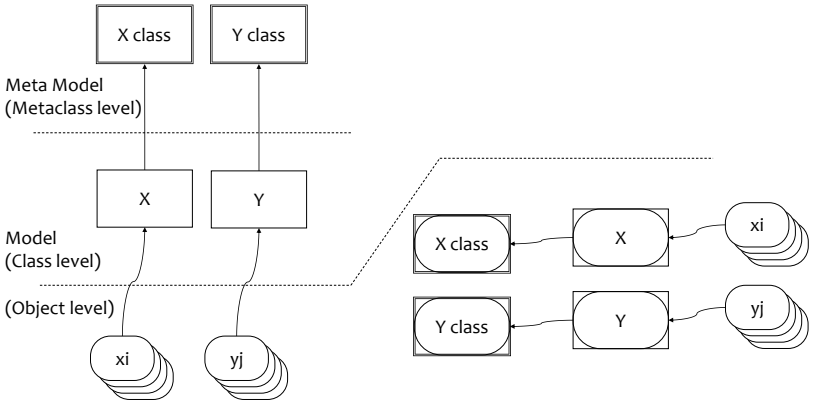

Figure 6: (left) the Smalltalk-80 Metaclass/Class/Object model as a three-layer model; (right) the implementation, all in the Object layer

2. Reifying an emergent class (for example, the Trail class in figure 5). The concept did not exist in the model (there is no relevant superclass), but does in the metamodel (once AggregateType is reified). The reified class is a new instance of that concept: the trail is a new kind of aggregate object, a new kind of thing with new kinds of behaviours, roughly analogous to a new species or genus in biology.

3. Reifying an emergent metaclass (for example, the aggregate type in figure 4). The concept did not exist in the metamodel: the aggregate type is a new kind of meta-object, a new concept in the language, roughly analogous to a major transition in evolutionary biology [17] (for example, the move from unicellular to multicellular organisms).

Such reification provides the requisite novelty generating power, when implemented in a computational system.

Dynamic Models and Metamodels

The process of changing the model and metamodel needs to be dynamic, so that we can add reified emergent classes and metaclasses as they emerge and are recognised at runtime. Smalltalk-80 [7] provides an approach to this. Two fundamental concepts in Smalltalk-80 are: everything is an object; an object is an instance of some class. Since everything is an object, a class is an object, and so is an instance of some class, called its metaclass. So object x is an instance of class X, and class X is the (singleton) instance of its metaclass, referred to[4] as X class. Since everything is an object, a metaclass is an object, and so is an instance of some class, the class Metaclass[5].

So Smalltalk-80 has the objects, the classes (model) and metaclasses (metamodel) *all available as objects at runtime* (figure 6). All can be instantiated, deleted, and modified at runtime, via this *computational reflection* ("a reflective system is a computational system which is about itself in a causally connected way" [16]). Although Smalltalk-80 is not a pure reflective language, it does have reflective capabilities, and many others can be added programatically [6].

Other computationally reflective languages (ones that can modify themselves at run-time, to a greater or lesser extent) include Lisp, Prolog, Python, Ruby, and JavaScript.

Examples of self-modifying and reflective systems

Suber [23] discusses self-amendment in the context of law making, and describes Nomic [10][23, appx.3], a (non-computer-based) law-based game where changing the rules (including the rule that players must obey the rules) is a move. Suber asks if it is possible either to make some rules unchangeable whilst preserving the power to amend others, or to irrevocably repeal the power to amend the rules.

Reflection is key in the branch of Artificial Intelligence concerned with "learning to learn", metamemory and metacognition [4, 14, 15, 18, 21, 24]. Learning changes the model; learning how to learn, learning a better learning algorithm, is changing the metamodel. Note that our concern here is not in high-level *cognition*, however, but in the role of reflection in open-ended *evolution*.

Biology is the ultimate self-modifying system. Hickinnbotham et al [9] describe a self-modifying computational

[4] In Smalltalk-80, metaclasses are not explicitly named. A metaclass can be referred to by sending the message class to the class's single instance. The value of this message expression is the metaclass. So the metaclass of class X can be referred to as X class. (Since there is also a class called Class, this terminology can lead to awkward constructions, such as "the class Class class".)

[5] Of course, since Metaclass is a class, it is the singleton instance of its metaclass, Metaclass class. And Metaclass class is a metaclass, so like all metaclasses, it is an instance of Metaclass. This circularity stops the potential infinite regress of needing meta-metaclasses, etc. See [7, pp268-72] for details.

architecture inspired by biological DNA, RNA and protein machines. Tomita et al [25] use graph-rewriting automata with five kinds of rewriting rules, to implement self-replication. They discuss the possibility of embedding the graph rewriting program as a graph itself within the system, allowing for execution to modify which rules are applied. This is analogous to modifying the model at run-time; an analogy to modifying the metamodel would be to introduce new kinds of rewriting rules.

Reflection is proposed as the route to self-adaptive software systems [1, 16]. The architectural requirements specified in [1] differ from our own here, however, because the application domain is very different. For example, [1] is concerned with reflection on programming language concepts, subject to real world domain constraints; we are concerned with reflection on novelty generating mechanisms, and need to impose constraints in terms of some energy model (next). They are concerned with software engineering structuring, clear separation of model and metamodel layers and their respective concerns, and with performance; we are concerned with open ended novelty generation, and embrace the biologically-inspired 'messiness' of deliberately mixing layers of abstraction. Consequently, they carefully separate domain and reflective aspects, and keep the computation to do with reflection in the metamodel level only; our architecture of computation is orthogonal to the model and metamodel layers (next), to enable reflection at all levels, not only the metamodel reflecting on the model.

An open-ended architecture

As discussed above, computational reflection provides a route to open-ended novelty. As Maes [16] says: "A language with reflective facilities is **open-ended**: reflection makes it possible to make (local) specialised interpreters of the language, from within the language itself."

Reflection provides the computational mechanism, but we also need an architecture within which to generate and run the open-ended code. Here we describe an architecture for such a system. We use OO terminology; this specific paradigm, although well-suited, is not necessary for the architecture, just some analogue of the underlying concepts in a reflective programming language.

We define only a bootstrap architecture. The whole point of computational open-ended novelty generation is for the system to modify this architecture at run-time.

The key feature is that the three levels – instance, class, and metaclass – all exist as executing and modifiable objects in the system at run-time. For the bootstrap, we separate the system into three subsystems: an initial seed application, the observer-reifier-modifier (ORM) intentionaliser, and the virtual machine (VM). The seed application, for example, some agent-based simulation, acts as the raw material from which the open ended novelty grows. The other two subsystems are described below. See figure 7.

ORM Intentionaliser: modifying the models

Our framework for intensionalising emergent structures has three components:

1. emergence **observers**, that observe novel emergent structures and behaviours
2. emergence **reifiers**, that intensionalise the recognised types, and add the relevant classes or metaclasses into the run-time, thereby changing the model or metamodel
3. model **modifiers**, that modify the simulation (instances, classes, or metaclasses) to exploit the reified structures

In [1], a distinction is made between structural reflection (reification of structural aspects such as data types) and behavioural reflection (reification of computations and their behaviours). It is crucial that emergence recognisers capture patterns both of structure and of behaviour: at different levels of emergence features can appear to be either 'particles' or 'processes' [22].

The ORM subsystem therefore includes ObserverType, ReifierType, and ModifierType metaclasses, and bootstrap class instances of these, to provide the meta-functionality. For example, we might have the class Eye as a bootstrap instance of ObserverType, whose own instances observe the simulation for particular spatial and temporal patterns that indicate emergence. An Eye instance might detect a flock- or trail-like emergent. It notifies a suitable Reifier instance, which can appropriately intensionalise the emergent, for example, as an internally detectable object. A suitable Modifier instance then modifies other classes in the simulation so that their instances can detect the new objects. It might also modify their behaviours to use the detected information, or, in an evolutionary simulation, allow these modified behaviours to evolve.

Key to the overall architecture is the fact that the simulator is reflective, not just at the core agent level, but throughout. Hence a bootstrap observer (for example) can observe not only novel agent patterns, but also novel observation, reification, and modification patterns, which can then be reified and modified appropriately. We bootstrap with Hammer and Eye classes; later Spanner and Ear classes can emerge and be reified. Eventually new ModifierType metaclasses could be reified. Hence the simulation can not only change itself, it can change the way it changes itself (this does imply requirements on the representation of modifier rules [14]).

Being able to modify the modifier, being able to produce new kinds of ways of recognising, reifying and modifying the simulation, closes the self-referential loop, and produces a truly open-ended system.

Virtual machine: constraints

The virtual machine provides whatever run-time support is needed for the ORM architecture, in the usual manner (at

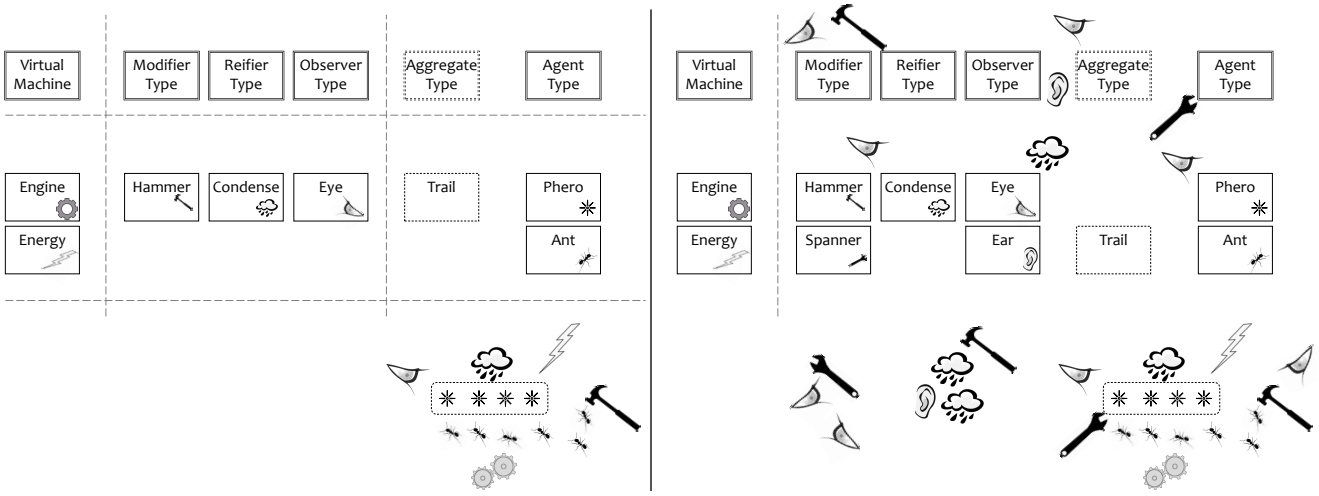

Figure 7: The architecture, showing metamodel, model (class boxes show an instance icon), and instance layers (horizontal dashed lines), and VM, ORM, and ABM subsystems (vertical dashed lines). (left) Minimal self-modification: the layers and subsystems are well-defined, the bootstrap ORM objects observe and reify the emergent trail class and modify the ABM objects, the ORM model and metamodel are fixed. (right) Constrained self-modification: the ORM components observe and modify the ABM and ORM objects, model, and metamodel (but not the VM), reifying emergent ORM components, and potentially modifying the kind of modifiers.

a minimum, compilation, dynamic object communication, and error handling). In addition, it provides some form of constraint on the modification processes. The research challenge is to achieve framework behaviour that allows a simulation to exploit emergent novelty without dissolving into chaos. A completely unconstrained framework could well modify itself out of existence. Some form of constraint, for example an analogue of conservation of energy, might be needed to allow the system to develop in interesting directions without devolving into a mess of object soup.

However, a completely constrained system, that allows no modification, no intensionalisation, is static and cannot achieve open-ended dynamics. This is the state of most classic ABM simulations.

It seems plausible that some degree of constraint between a totally static mode and meta model, and total freedom, is required; this is possibly some "edge of chaos" [13] requirement. Hence the role of the constraint is to help the system self-organise to maximally complex patterns of structure and behaviour.

Modifying the VM

If the virtual machine is implemented in the same language and at the same level as ORM, it could potentially also be a target of the self-modification process. Here we assume that the constraint part is to be unmodifiable, for the reasons given above, but the interpreter or message handler part is a valid target of modification.

Consider a Smalltalk-80 implementation. The simulation and modifier objects are Smalltalk-80 objects, and are implemented (given their execution semantics) in a Smalltalk-80 VM. A suitably defined physics engine could be included at the object level, and be subject to the same modification processes as the objects themselves.

Discussion

Consideration of a metamodel of emergence has led to the insight that emergent properties are emergent intensional definitions. The difference exhibits itself in simulations, where the emergent properties are observed via instrumentation, rather than reified directly. If the emergent properties are reified and intensionalised, with their own definitions and behaviours, they can become the kind of agents that result in (further) emergent properties.

In order for these kinds of emergent innovation to be included in a simulation, the simulation needs to be able to modify its own model, and metamodel, dynamically (at run time). We contend that **for a simulation to exhibit open ended dynamics, it must include a form of computational reflection** that allows it to modify its own model and metamodel as the simulation is running.

We have specified the design of an open-ended architecture. (The next stage of work is to develop a prototype implementation.) This architecture has the instances, model, and metamodel all available for modification at run-time. It has three subsystems: a virtual machine providing run-time support and modification constraints, an observer-reifier-modifier intensionaliser, and a seed application. This is a *bootstrap* architecture: successful self-modification will modify this architecture.

Rosen [19, §10a] argues that the difference between an organism and a mechanism is that an organism "is closed to efficient causation", and that a mechanism cannot be so closed. He uses Aristotle's term "efficient cause" as the cause that brings something about. He argues that life is self-defining, self-causing, autopoietic; but that simulations cannot be, that simulations require something outside the system to define them. We claim that the reflective approach and bootstrap architecture described above *can* allow simulations to be similarly self-defining, self-generating, self-causal, and hence to exhibit some of the properties Rosen requires for life.

Acknowledgments

Out thanks to Paul Andrews, Ed Clark, Tim Clarke, Simon Hickinbotham, Adam Nellis, Mungo Pay, Fiona Polack, Adam Sampson, Jon Timmis, Emma Uprichard, and Peter Young, for helpful discussions, and to the anonymous referees for their suggestions.

The work described here is part of the CoSMoS[6] project, funded by EPSRC grant EP/E053505/1 and a Microsoft Research Europe PhD studentship.

References

[1] Jesper Andersson, Rogerio de Lemos, Sam Malek, and Danny Weyns. Reflecting on self-adaptive software systems. In *SEAMS'09*, pages 38–47. IEEE, 2009.

[2] Mark A. Bedau. The nature of life. In Margaret A. Boden, editor, *The Philosophy of Artificial Life*. Oxford University Press, 1996.

[3] Donald T. Campbell. 'Downward Causation' in hierarchically organised biological systems. In Francisco Jose Ayala and Theodosius Dobzhansky, editors, *Studies in the Philosophy of Biology: reduction and related problems*, chapter 11, pages 179–186. Macmillan, 1974.

[4] Michael T. Cox. Metacognition in computation: A selected research review. *Artificial Intelligence*, 169(2):104–141, 2005.

[5] James P. Crutchfield. The calculi of emergence. *Physica D*, 75:11–54, 1994.

[6] Brian Foote and Ralph E. Johnson. Reflective facilities in Smalltalk-80. In *OOPSLA'89*, pages 327–335. ACM Press, 1989.

[7] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[8] Simon Hickinbotham, Edward Clark, Susan Stepney, Tim Clarke, Adam Nellis, Mungo Pay, and Peter Young. Diversity from a monoculture: effects of mutation-on-copy in a string-based artificial chemistry. In *ALife XII*, pages 24–31. MIT Press, 2010.

[9] Simon Hickinbotham, Susan Stepney, Adam Nellis, Tim Clarke, Edward Clark, Mungo Pay, and Peter Young. Embodied genomes and metaprogramming. In *ECAL 2011*. MIT Press, 2011.

[10] Douglas Hofstadter. Metamagical themas. *Scientific American*, June 1982.

[11] Dominic Hyde. Sorites paradox. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2008 edition, 2008. http://plato.stanford.edu/archives/fall2008/entries/sorites-paradox/.

[12] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: the Model Driven Architecture: practice and promise*. Addison-Wesley, 2003.

[13] Chris G. Langton. Computation at the edge of chaos: phase transitions and emergent computation. *Physica D*, 42:12–37, 1990.

[14] Douglas B. Lenat and John Seely Brown. Why AM and EURISKO appear to work. *Artificial Intelligence*, 23:269–294, 1984.

[15] Luís Seabra Lopes and Aneesh Chauhan. Open-ended category learning for language acquisition. *Connection Science*, 20(4):277–297, 2008.

[16] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA'87*, pages 147–155. ACM Press, 1987.

[17] John Maynard Smith and Eörs Szathmáry. *The Major Transitions in Evolution*. Oxford University Press, 1995.

[18] Thomas O. Nelson. Metamemory: A theoretical framework and new findings. *Psychology of Learning and Motivation*, 26:125–173, 1990.

[19] Robert Rosen. *Life Itself*. Columbia University Press, 1991.

[20] Kepa Ruiz-Mirazo, Jon Umerez, and Alvaro Moreno. Enabling conditions for open-ended evolution. *Biology and Philosophy*, 23:67–85, 2008.

[21] Jürgen Schmidhuber, Jieyu Zhao, and Nicol N. Schraudolph. Reinforcement learning with self-modifying policies. In Thrun and Pratt [24], pages 293–309.

[22] Susan Stepney, Fiona Polack, and Heather Turner. Engineering emergence. In *ICECCS 2006*, pages 89–97. IEEE, 2006.

[23] Peter Suber. *The Paradox of Self-Amendment: a study of law, logic, omnipotence, and change*. Peter Lang, 1990. http://www.earlham.edu/~peters/writing/psa/.

[24] Sebastian Thrun and Lorien Y. Pratt, editors. *Learning to Learn*. Kluwer, 1997.

[25] Kohji Tomita, Satoshi Murata, and Haruhisa Kurokawa. Self-description for construction and computation on graph-rewriting automata. *Artificial Life*, 13(4):383–396, 2007.

[6] http://www.cosmos-research.org