Embodied genomes and metaprogramming

Simon Hickinbotham[1], Susan Stepney[1], Adam Nellis[1],
Tim Clarke[1], Ed Clark[1], Mungo Pay[1], Peter Young[1]

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

Abstract

We model some of the crucial properties of biological novelty generation, and abstract these out into minimal requirements for an ALife system that exhibits constant novelty generation (open ended evolution) combined with robustness.

The requirements are an embodied genome that supports run-time metaprogramming ('self modifying code'), generation of multiple behaviours expressible as interfaces, and specialisation via (implicit or explicit) removal of interfaces.

The main application of self modifying code to date has been top down, in the branch of Artificial Intelligence concerned with *learning to learn*. However, here we take the bottom up Artificial Life philosophy seriously, and apply the concept to low level behaviours, in order to develop emergent novelty.

Introduction

It is proving very hard to develop *in silico* ALife systems that exhibit open-ended novelty generation. This may be because many such systems are *closed* in that they often have pre-designed and fixed algorithms, and fixed information representations. The scope for these systems to generate novelty is heavily constrained by these design decisions. This closure is in sharp contrast to biology, where its 'algorithms' and 'representations' are themselves products of the novelty generation processes.

In this paper, we go back to biology, and look at certain aspects of its processes that are key to its power to generate novelty. We use these to develop an open computational novelty generation architecture.

A key source of open-ended biological novelty seems to be the embodiment of the genome in a form that makes it accessible to the other active elements of the system: the DNA can be modified by proteins, changing what future proteins are expressed, and what future modifications occur.

We propose that an analogous approach is needed for open-ended computational innovation. The 'computational DNA' (program code) must be accessible to and modifiable by the active elements (executing program). This can be achieved through *run-time metaprogramming*. (Metaprogramming is when programs manipulate programs; here the manipulator and manipulated are the *same* program, and the manipulations are performed at run-time. This is also known as reflective programming in high level languages, and as self-modifying code in assembly languages.)

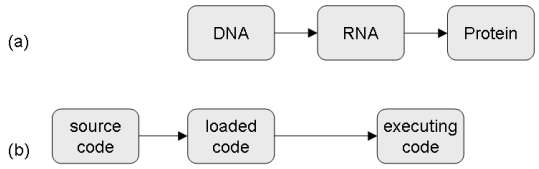

Figure 1: (a) Information flow in the central dogma of molecular biology; (b) control flow in classical computer programs. The vertical alignments indicate rough analogy, discussed later.

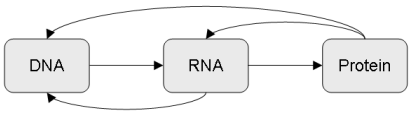

Figure 2: Control flow in the cell

Biological models

Self-innovation circular architecture

Crick's central dogma of molecular biology, first stated in 1958 [5], has a linear flow of information content (DNA → RNA → protein, figure 1a). This informational statement is often more strongly interpreted to mean a linear *control pathway*, with DNA 'in control' of the system, and no returning control paths. The standard paradigm of computation has an analogous flow of control (source code → loaded code → executing code). The source code is 'in control'; all subsequent events are a direct consequence of this code (figure 1b).

Such linear flow models are a simple way to describe causality in a system. However, the linear flow of *control* in biology is false. Proteins act on the RNA, and both RNA and proteins act on the DNA, controlling what is expressed, and even changing the DNA (figure 2). There is no strict

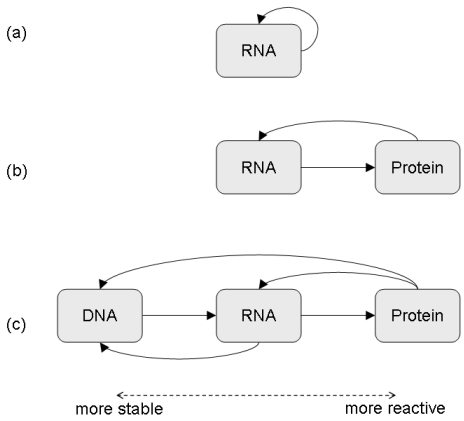

Figure 3: (a) original RNA-world; (b) RNAs and proteins; (c) today's DNA, RNA and protein world. Solid arrows represent the direction of control flow and effect; dashed line represents a molecular stability spectrum.

linear flow of control; it is a closed loop with all entity types able to affect all other entity types. This circularity of interaction allows the emergent biological properties, including novelty generation.

We propose that an analogous circularity of interaction is needed for computational novelty generation: self-modifying self-producing computer *code*, achieved through *run-time metaprogramming*.

A history of specialisation

Prebiotically there were molecules. Novelty generation resulted in molecules with additional behaviours: RNA encodes information, and can use that information in two ways, as an active *machine*, or as a passive *template*.

In RNA-world [12, 30], the information-bearing template and the active machinery are the same kind of molecules: RNA. However, these two behaviours require different kinds of properties: information-bearing templates require relative stability, whereas the machinery requires reactivity. Biology's solution was to specialise with two sets of molecules: RNA (mainly) for information, and proteins for reactivity. This specialisation continued until today's situation, with the even more stable DNA providing long-term stability for information storage. (See figure 3.)

The phenotype of the genome

DNA expresses proteins. DNA is composed of *nucleotide bases*; proteins are composed of *amino acids*. These have different reactivity, yet they interact with each other in a variety of ways, such as chemical binding and topological entwining. In particular, different portions of the DNA are physically inaccessible at different stages of the cell cycle. These interactions are subject to selection pressures (limited by physico-chemical constraints), which has led to the emergence of important biological properties, such as: mutating at differing rates for different genes; specifying when genes express proteins and at what rates; organising the co-location of genes for particular metabolic pathways. These are components of biological innovation.

Analogous properties are not seen to *emerge* in computer simulations (although they can be explicitly designed in).

In order to build a computational analogue of the relevant biological processes, we need to carefully distinguish the genome and the DNA/RNA: (a) the genome is an *abstraction*, a sequence of codes; (b) the DNA (or RNA, in RNA-world) is a physical molecule, *embodying* the genomic information. A protein is another class of physical molecule, its sequence encoded by the genome, physically expressed from the DNA/RNA. In many models of biological evolution the genome and the DNA/RNA that it represents are taken to be synonymous, and the DNA/RNA is modelled differently from the proteins. In reality, however, the genome is an abstraction, and is a different category from the DNA/RNA molecule that is the physical embodiment of that abstraction. The DNA/RNA is an intrinsic part of the *phenotype* of the organism, of the same category as the proteins. Being embodied, it interacts with, and is acted on by, enzymes and metabolites (though less readily than the other entities in the cell).

This embodiment, which we hypothesise is necessary for biological novelty generation, provides the inspiration for our computational architecture to produce analogous open-ended novelty generation *in silico*.

Computational analogues

We take this aspect of biology, of circular interaction enabled by an embodied template, as inspiration for the design of a computational form of novelty generation.

We perform the following process [27] (for two related biological systems, RNA-world and DNA-world): we produce a model of the *biological* system; we abstract this into a *conceptual* model of the underlying processes and relationships (not shown here); we *instantiate* the conceptual model in computational terms. We use UML class diagrams to express these models.

AChems as analogues of RNA-world

We first look at the simpler RNA-world (figure 4a). Physics determines how molecules interact, through features such as molecular folding and binding affinities. The genome is an abstraction of the information in the RNA. The biological RNA is embodied: RNA molecules express and are modified by RNA molecules.

The computational analogy is self-modifying code (figure 4b). The analogue of the (disembodied) genome is the (disembodied) source code. The analogue of the active RNA is the executing code: for the analogy to hold, the executing code must be able to modify its own instructions. The

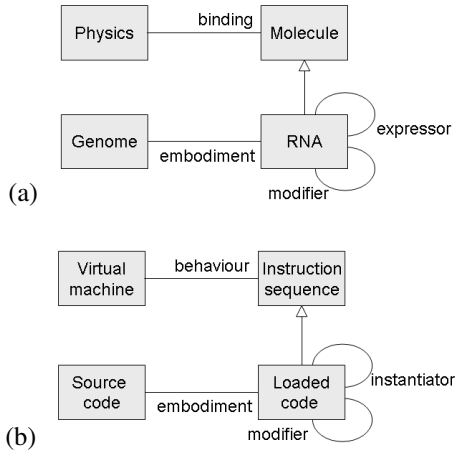

Figure 4: UML class diagram of RNA-world: (a) biological model of embodied RNA; (b) conceptual model instantiated with an AChem

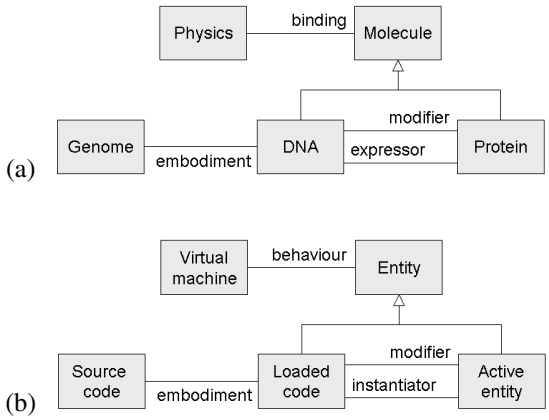

Figure 5: UML class diagram of DNA-world: (a) biological model of embodied DNA and protein machine (omitting the role of RNA, for emphasis); (b) conceptual model instantiated with metaprogramming

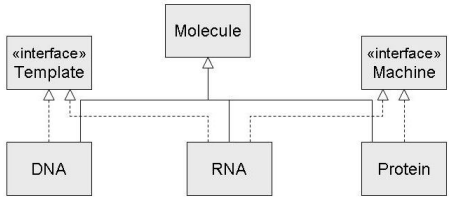

Figure 6: UML class diagram of molecules realising (behavioural) interfaces

analogue of the physics (which defines how molecules can interact) is the virtual machine (which describes the AChem program language semantics, and how the various AChem objects can interact).

An assembly language level Artificial Chemistry, where the executing code is able to modify the instructions ('embodied source code'), provides a computational model here. Examples include Tierra [26], Avida [1], and stringmol [15, 14, 16], where the 'chemicals' are direct analogues of the RNA strands.

Reflection as an analogue of DNA-world

Many modern high-level programming languages are designed to enforce a strict separation between code and data, and cannot self-modify in this way. But not all.

We next look at 'DNA-world', a biologically later specialisation of RNA-world (figure 5a). The biological DNA is embodied, and is affected and modified by the proteins it expresses. (Notice this model does not make the biological role of RNA explicit in this process. Here we wish to emphasise the distinction between stable information archive and active machine, so we abstract these as 'DNA' and 'protein' respectively, and omit the intermediate RNA for the purposes of our argument.)

Analogously, the computer source code is *embodied*, and affected and modified by the executing code it specifies (figure 5b). Here we need a programming language where there is a separation between code-representing entities and other active entities (unlike in the RNA/assembly language analogy) that can nevertheless interact at run-time. A high-level language with computational reflection [24] is suitable here: the source code is embodied in the run-time system, and can be modified by the executing system, but is (conceptually) separate from it.

Smalltalk-80 [13] is a good example. In Smalltalk, the source code is just another data structure that can be manipulated by the executing program. Smalltalk is a pure object-oriented language: every value is an object, including classes and code blocks. Code blocks, including ones that modify and create classes, can be constructed at run-time and then executed. An executing Smalltalk system thus has the ability to modify and extend itself: its source code is embodied in the executing system.

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.

Novelty versus specialisation

RNA encodes information, and can use that information in two ways, as a passive *template*, or expressed as an active *machine*. We can model these two different uses in UML as *interfaces* (figure 6). The interfaces capture the specific behaviours exhibited by certain molecules.

Later (in the RNA-world model), *specialisation* occurred. Molecules that had only one of these behaviours, either machine (protein) or template (DNA), emerged. Once specialised components (components that have *lost* an interface) have emerged, they can adapt to perform their specialisation (remaining interface) more effectively. (Biologically,

specialisation to DNA templates and protein machines was mediated by RNA, and there are still RNA molecules that can be interpreted as fragments of this mediation in modern organisms [3].)

So in modelling terms, novelty generation is *creation* of new interfaces, specialisation is *removal* of interfaces from sub-species of agents. In code terms, removal of an explicit interface is simple: it is just deleted. However, removal of an implicit, emergent interface is not so simple: the low-level behaviour has to change such that the interface behaviour no longer emerges. This is the case both in molecular terms, and in low-level AChem systems. In the molecular case discussed here, this process occurred through differentiation into molecules with distinct chemical structures (nucleotide bases in DNA versus amino acids in proteins). This necessitated the introduction of a *decoding* element to the expression relation, to translate from one structure to the other.

We suggest that such a differentiation step will be helpful in any analogous AChem system designed to progress beyond RNA-world level behaviour in this manner. Specialisation of template and machine behaviours requires templates to be less reactive and machines more reactive. Although such differentiation may be achieved in a homogeneous system (for example, by altering ratios of symbols in the underlying alphabet), it is is made easier by having some *structural* difference between them, to help this *behavioural* difference emerge. Given a structural difference, translation will be required to take the template into its machine expression. This translation requirement is not inconsistent with template and machinery being the same kind of thing. There is sufficient richness in chemistry to allow DNA and proteins to be the same kind of thing (molecules) whilst having different representations of their information content (nucleotide bases versus amino acids). The similar form of embodiment allows the information to be modifiable by the system it encodes, whilst the different representations provide the separation of properties that help support specialised behaviour. Chemistry is rich enough to provide this spectrum: AChems will need analogous richness.

High level languages can provide explicit support for this process of specialisation. For example, one pattern supported by refactoring tools is *Extract Interface* [11, p.341]. Aspect oriented programming [18] allows particular kinds of behaviour to cut across the code structure. These are both design time, rather than run time, processes, but some of the concepts may be automatable. Another concept, relevant to implicitly-defined interfaces, is duck typing [20], which allows the type to be determined dynamically, based on what methods a class currently supports.

The 'softness' in losing (specialising away) an interface is an important property in terms of robustness through redundancy and degeneracy. It is not necessary for a specialist to lose an interface completely, only for the system to lose reliance in it on providing the interface. The specialist can safely modify other things about itself, but it might still maintain some ability to implement some part of the interface in an 'emergency'. If enough parts of the system can implement parts of the interface adequately, then this degeneracy amounts to the system as a whole implementing the whole interface. This provides a form of distributed backup, in case of failure of the machine that is 'supposed' to implement the interface.

Computational architecture

The previous discussion leads us to the notion that, to get emergent novelty in simulation, we should look to run-time metaprogramming. In such a system the code has never finished being written, so the program cannot finish running. Open ended computation is obtained, allowing unprescribed novelty generation within the computer. The main application of self modifying code to date has been top down, in the branch of Artificial Intelligence concerned with *learning to learn* [22, 28, 29]. However, here we take the *bottom up* Artificial Life philosophy seriously, and apply the concept to *low level behaviours*, in order to develop emergent novelty.

Run-time metaprogramming on its own is not sufficient; we also need an architecture within which to run the code. The biological models above can help us here, too. There are two aspects to the architecture. One comes from the class box Physics in figures 4a and 5a, one from the roles modifier and expressor.

Physics engine

Underlying biology there is physics and chemistry: the processes that define how molecules move around, how they can interact (for example, binding affinities), what the result of the reaction is, and the constraints on the system (for example, conservation laws). In an artificial system, we have to explicitly implement analogues of many of these processes. The usual way to do this is in terms of a *virtual machine* (VM), often referred to as a 'physics engine', that provides the execution environment in which the molecule-analogues exist. Tierra [26], for example, has an explicit VM that executes the Tierra assembly language.

The first point to note is that physics is **uncrashable**: there is no real world analogue of a computational core dump or fatal exception. There are two ways to achieve this in the computational architecture: language design or VM handling. The molecular language can be designed such that any molecular interaction results in a legal behaviour. This is relatively straightforward at the assembly language level (care still has to be taken not to access areas outside legal memory). Alternatively, the VM can be designed to trap and isolate any unhandled exceptions. For higher level languages that are modifying themselves, this will become the necessary route.

Next, the VM provides the **spatial dynamics**: how the entities move around, and so who can interact with whom.

This can be explicitly spatial, or be a 'well mixed' aspatial model, or even a hybrid (a spatial arrangement of containers with aspatial contents, for example).

We want a system that can generate open-ended novelty without dissolving into chaos. A completely unconstrained system could well modify itself out of existence. Some form of constraint might be needed to allow the system to develop in interesting directions without devolving into a mess of molecule soup. However, a completely constrained system, that allows no modification to its architecture and representations, is static and cannot achieve open-ended dynamics. This is the state of most classic agent-based simulations. The VM should provide such constraint through an **energy model**. This is some analogue of the constraints that real-world physics provides, such as conservation of energy. This provides a limited resource for the various entities; in particular, it prevents 'free copying', or unlimited replication, and so provides an evolutionary pressure [8, 17]. It is important not to have a 'closed' energy system, however: this would lead to equilibrium. Biological systems are far-from-equilibrium systems, maintained there by an energy *flux*. More sophisticated VMs might also provide an analogue of entropy.

It seems plausible that some degree of constraint between a totally static model, and total freedom, is required; this is possibly some *edge of chaos* [21] requirement. Hence the role of the constraint is to help the system self-organise to maximally complex patterns of structure and behaviour.

Some choices of what goes in the VM and what goes in the molecular language are design decisions. For example, it can be beneficial if the entities have a limited lifetime: this results in entities having to renew themselves to survive, which imposes a natural evolutionary pressure on the system. Whether such a **decay** process is implemented in the VM or in the entities themselves is a design decision: the choice will determine how much the decay can be affected by the intrinsic evolutionary process. The presence of such a decay mechanism has consequences. For example, it means that there will need to be multiple copies of certain machine templates (or templates need to have very different decay properties from active machine molecules), so that the decay of a template does not permanently lose a solution.

Modification and expression machines

The physics engine provides the VM within which entities can interact and generate novelty (novel entities, novel behaviours, novel interactions). We need some initial entities to set the system going.

Consider the roles modifier and expressor in figures 4a and 5a. In biology, these are embodied, 'implemented' by specific machine molecules (ribosomes, transposons, chaperone proteins, etc). Additionally, there are machine molecules that do things not related to self-modification: these are the active molecules performing the external 'function' of the system. This provides a route to embedding application-specific behaviours into a novelty generating architecture.

A novelty generating system could be *bootstrapped* with some specialist machines for these various tasks. this involves writing the bootstraps as code for the embodied templates that, when expressed, becomes the active machine. The key point is that these bootstrap machines *are all encoded on the template*, and so are themselves subject to modification, either directly, by a modifier machine changing their encoding, or through imprecise replication by a 'sloppy' replicator machine. And these various modification machines are themselves subject to modification. This is why we are describing only the 'bootstrap' architecture: the self-modification processes will then develop new machines, new kinds of machines, and new ways of expressing and otherwise generating machines. This self-modification is what breaks away from fixed algorithms and fixed representations, and allows open-ended novelty generation.

Different kinds of bootstrap machines are suggested by different stages of biological evolution. We could bootstrap with only replicator machines (machines that can copy templates). This is the approach we have taken in our original stringmol AChem [14, 15, 16]. Here we wish to short-circuit the process of evolving all novelty from scratch, but in a way that does not compromise further open-ended novelty generation. We can do so by bootstrapping the system with some more sophisticated machines, some inspired directly by the biological processes of figures 4a and 5a, and some higher level ones implementing 'non-atomic' functionality. There is a tension between performance (composing the actions of low level machines versus the single action of a ready-made higher level machine) and flexibility (being able to compose low level machines in novel ways, and having their modifications being more likely to produce viable variant machines). The aim is to engineer a sufficiently powerful and flexible bootstrap that the system can smoothly self-modify into an open ended novelty generator.

Candidate bootstrap machines (which would need to be designed both for the implementation language, and for any application) include those to perform the following functions:

- **expression**: a machine that takes a template, and expresses (instantiates) some machine encoded there. This does not need to be restricted to simple 'gene expression': some machines might use information in the template in different way, for example, analogous to the use of 'gene libraries' in assembling antibodies. The expression machine might be 'sloppy', expressing a range of similar machines, with this sloppiness subject to modification.

- **modification**: a machine that takes a template, and modifies its content in some language dependent way (possibilities include low level machines analogous to transposons

[10], retroviruses [2, 23], and F-plasmids [19], and higher level machines analogous to the processes of gene error correction and crossover, for example).

- **regulation**: a machine that regulates the action of expression machines (this is not explicitly included in the UML models above, but gene regulation is a known critical aspect of biological control, and the regulation is performed by machine-class molecules).
- **replication**: a machine that replicates templates. There will be a constant turnover of templates in RNA-world analogues, and a slower turnover in the more template-stable DNA-world analogues. The replication machine should be 'sloppy', providing a source of variation, with this sloppiness subject to modification.
- **translation/transduction**: machines that translate between different information-bearing formats (both internal, and input/output)
- **application**: machines that perform application-specific tasks (the analogue of protein machine behaviours that are not related to modification and expression)

As well as these directly biologically inspired machines, other 'higher-level' bootstrap machines might be developed, to help kick-start specific kinds of novelty generation. These are inspired by even later developments in biological evolution. Such machines might include:

- **sensors**: machines that can sense the internal state of the system (for example, via quorum sensing), which information may be used by transducers, regulators, etc
- **generators**: machines that write new templates based on observed behaviours in the system (for example, 'reverse engineering' the composed behaviour of several low level machines into a single high-level machine, or breaking down a high level machine into component behaviours)

Other application-specific bootstrap machines can be designed as required. Design of such machines needs to respect the architecture of the system, in particular, the 'soft' nature of the mechanisms [4], and the continual turnover of the machines (a good solution, once found, must then be maintained).

Some of these bootstrap machines (particularly higher-level ones) will be easier to implement in high level languages than in assembly-level AChems. However, they are constrained by the particular physics of the system. For example, if the system's physics does not support global observation, then a global observer machine will not be directly implementable in the system (however, a property akin to global observation could potentially *emerge*). Machines in high level languages can nevertheless be bootstrapped to have potentially sophisticated memories and behaviours. There is, however, a tension between the sophistication of the machine that allows it to perform complex functions, and the simplicity of the machine that allows it to be modified in useful ways. Any higher level bootstrap machines should be implemented as compositions of simpler machines wherever possible, allowing modification both of the machines themselves and the ways they are composed. That is, the *representation* of these machines should also be modifiable.

Biological messiness

Bio-inspired systems are abstractions of the myriad emergent phenomena seen in biology. Their goal is to develop toolsets that *efficiently* distil the unique properties of robustness and adaptability seen in biological systems. Care has to be taken not to throw the baby out with the bathwater, however. We propose that biology generates emergent phenomena by coupling together two phenomena. The first of these is massive *redundancy and degeneracy*, observable in many biological networks: entities are rarely the 'sole providers' of all their functionality. This generates massive 'baseline diversity'. The second is natural selection, which builds hierarchical emergent behaviours by reinforcing beneficial interactions. Crucially *diversity is maintained*, both within *and between* units of selection, allowing further interactions to be developed and built upon.

This messiness, redundancy and degeneracy that pervades biology has 'function', in that it provides a sort of embodied memory. It endows the system with robustness, and alternative pathways should the environment change. It is important not to simplify this away when building abstract models of the processes. In terms of the models introduced above, components should be allowed multiple interfaces, with different components realising different subsets of the complete set of interfaces.

Multiplicity and concentration of machines are an important part of this messiness. Many molecules need to exist in a concentration in order to collectively fulfil their role (DNA being the exception). Given the vast multiplicity of some molecules, 'erroneous' molecules that have partial functionality cannot be easily removed, if they do not result in the death of the organism before reproduction. Checking the viability of a molecular unit is an extremely expensive process in biology and is not normally attempted (DNA again being the exception). The continual decay and replenishment is the preferred mechanism. For example, the cell membrane is continually created and consumed [6], and there is a dynamic turnover of flagella motors [7].

This further suggests that there should be multiple copies of templates and machines in the computational system.

Comparison with existing systems

We are not aware of any high level reflective language systems that fit our DNA-world framework.

A good example of such a computational system that fits our RNA-world framework is an assembly language where the executing code is able to modify the instructions ('embodied source code'). For example, consider an Artificial Chemistry such as Tierra [26], Avida [1], or stringmol [14, 15, 16], which take approaches that are direct analogues of RNA-world. Their chemicals affect and modify each other, by the computational execution of the AChem. However, none fits all the requirements of our framework.

Tierra, directly inspired by RNA-world, fits quite closely with part of our architecture, but has two major differences.

Tierra has an explicit VM to execute its assembly language, designed to be "especially hospitable to synthetic life": non-brittle and evolvable. The spatial model is provided by location in computer memory (although instructions can point to anywhere in space). The entities are analogues of "creatures of the RNA world", although the individual machine instructions are considered to be more analogous to the more chemically active amino acids than to RNA's nucleotide bases. Tierra uses CPU time-slices as an analogue of energy, with the size of the time slice being a tunable function of the entity's size: small size can be rewarded, discouraging 'bloat', or large size can be rewarded, encouraging complexity. It has a decay mechanism in the VM: killing entities when the memory space is close to full. The code can generate errors, which are used to increase the probability of the offending entity being killed. Sloppiness is hardcoded in the VM as bit-flip mutation rates (a background rate, and a higher rate on copy) [9], and through flawed instruction execution. The system is initialised with a single hand-crafted self-replicating entity.

Tierra does not fit our architecture in two important ways.

Firstly, and most importantly, although entities can read and execute the code of other entities, they can modify only themselves (each entity's memory space is write protected). This disallows the emergence of a population of mutually self-modifying entities, other than by copying foreign code into the host entity (a 'pull', rather than a 'push', mechanism). It is a model of single active machines, not of mutually interacting machines mutually defining their properties. This design decision, along with making a less 'brittle' programming language, was made with the aim of overcoming problems in earlier 'Core Wars' implementations (eg, [25]), where mutations mostly just destroyed the system. We believe that the biological inspiration strongly supports mutual modification, however, and that the routes to overcoming the Core Wars issues are a more sophisticated energy model, and a 'softer' language, particularly in respect to binding properties [4].

Secondly, the Tierra energy model is limited. There is no analogue of an energy store (battery, fat reserves) that would enable entities to 'time-shift' their use of the resource, or hand on a surplus to their progeny; Tierra is a 'use it or lose it' model. (Ray [26] mentions a possible extension allowing capture of CPU slices.) Nevertheless, Tierra evolves an interesting diversity of entities, particularly a range of parasite types.

Avida, although directly inspired by Tierra in the sense that it is an assembly-language based AChem using CPU time slices as a selection pressure, has a very different architecture and motivation from our approach. Entities, in fixed locations in 2D space, interact only with their neighbours, and then only through replication, which copies the replicated entity over its oldest neighbour. Bonus time slices, which can accumulate, are used as an explicit reward mechanism to evolve entities to perform certain tasks.

Stringmol is an assembly language AChem that fits our architecture quite closely, but not perfectly. It is a 'soft' replicator system that has generated novel emergent macromutations and hypercycles (two co-dependent species that replicate each other, but are not self-maintaining) [4, 14]. Its execution model involves two strings, and active machine and a passive template; however execution can change either string. The system is initialised with multiple copies of a hand-crafted replication machine, that can replicate any template string it binds to. We have not investigated its behaviour with other kinds of bootstrap machines.

Stringmol has an explicit energy model, in that a certain number of units are added to the container at each timestep, and molecules need to use an amount to execute each instruction. Hence there is a pressure to be small, to enable faster replication cycles. However, the energy is a global resource (energy is not stored in individual entities, but in the system and accessible to all). This removes any incentive for an individual entity to be frugal (beyond replication speed); stringmol exhibits the 'free rider' problem.

Summary and Conclusions

Biology uses a variety of processes to generate novelty and robustness. Fundamental is the capture of genomic information in an *embodied* genome (DNA or RNA) that is the same kind of structure (molecule) as the active machinery (RNA or proteins). This embodiment allows the active structures to interact with, control, and *modify* the information that defines them. Once novelty has been generated, it can be specialised into different components (DNA as information template, protein as active machine), allowing more effective behaviours to evolve, as the competing requirements of different behaviours are isolated in different components. Specialisation of template and active machinery is aided by different representations (at some level), which require a translation step from information encoded in the template to its expression in the machinery. Specialisation should not go too far: degeneracy and redundancy are also crucial components of biological robustness and adaptability.

Taking these concepts, and abstracting them, we can develop a set of requirements for analogous AChem and ALife implementations: (1) run-time metaprogramming, where

the executing system changes the program that defines its execution, including novelty generation as addition of interfaces; (2) a physics engine VM; (3) specialisation in terms of removal of interfaces (either explicitly, or implicitly by separation of implementation structure); (4) an expression step that decodes information on the template into a different representation on the machine (allowing different kinds of behaviour); (5) redundancy and degeneracy in terms of allowing multiple interfaces per component, and multiple copies of components; (6) sufficiently sophisticated bootstrap machines to short-circuit the origin of life process.

We claim that a suitably 'rich' computational environment based on an embodied, modifiable genome that allows novelty generation (adding interfaces) and specialisation (removing interfaces) is a necessary component in maintaining diversity and producing novelty.

Acknowledgments

This work is part of Plazzmid, EPSRC grant EP/F031033/1. Thanks to Alastair Droop and Tim Hoverd for helpful discussions, and to the anonymous referees for their helpful comments.

References

[1] C. Adami, C. T. Brown, and W. Kellogg. Evolutionary learning in the 2D artificial life system "Avida". In *Artificial Life IV*, pages 377–381. MIT Press, 1994.

[2] J. M. Bishop. Cellular oncogenes and retroviruses. *Ann. Rev. Biochem.*, 52:301–354, 1983.

[3] T. R. Cech. Exploring the new RNA world, 2004. http://nobelprize.org/nobel_prizes/chemistry/laureates/1989/cech-article.html.

[4] E. Clark, A. Nellis, S. Hickinbotham, S. Stepney, T. Clarke, M. Pay, and P. Young. Degeneracy enriches artificial chemistry binding systems. In *ECAL 2011*. MIT Press, 2011.

[5] F. Crick. Central dogma of molecular biology. *Nature*, 227:561–563, 1970.

[6] E. A. Dawidowicz. Dynamics of membrane lipid metabolism and turnover. *Ann. Rev. Biochem.*, 56(1):43–57, 1987.

[7] N. Delalez and J. P. Armitage. Parts exchange: tuning the flagellar motor to fit the conditions. *Mol. Microbiol.*, 71(4):807–10, 2009.

[8] P. S. di Fenizio. A less abstract artificial chemistry. In *Artificial Life VII*, pages 49–53. MIT Press, 2000.

[9] A. P. Droop and S. J. Hickinbotham. Application of small-world mutation topologies to an artificial life system. In *ECAL 2011*. MIT Press, 2011.

[10] C. Feschotte and E. J. Pritham. DNA transposons and the evolution of eukaryotic genomes. *Ann. Rev. Genetics*, 41:331–368, 2007.

[11] M. Fowler. *Refactoring*. Addison-Wesley, 1999.

[12] R. F. Gesteland, T. R. Cech, and J. F. Atkins. *The RNA World*. Cold Spring Harbor Press, 3rd edition, 2005.

[13] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[14] S. Hickinbotham, E. Clark, S. Stepney, T. Clarke, A. Nellis, M. Pay, and P. 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.

[15] S. Hickinbotham, E. Clark, S. Stepney, T. Clarke, A. Nellis, M. Pay, and P. Young. Specification of the stringmol chemical programming language v 0.1. Technical Report YCS-2010-457, University of York, 2010.

[16] S. Hickinbotham, E. Clark, S. Stepney, T. Clarke, A. Nellis, M. Pay, and P. Young. Molecular microprograms. In *ECAL 2009*, volume 5777 of *LNCS*, pages 291–298. Springer, 2011.

[17] T. Hoverd and S. Stepney. Energy as a driver of diversity in open-ended evolution. In *ECAL 2011*. MIT Press, 2011.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[19] B. C. Kline. A review of mini-F plasmid maintenance. *Plasmid*, 14:1–16, 1983.

[20] A. Koenig and B. E. Moo. Templates and duck typing. *Dr. Dobbs*, June 2005.

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

[22] D. B. Lenat and J. S. Brown. Why AM and EURISKO appear to work. *Artificial Intelligence*, 23:269–294, 1984.

[23] R. Lower, J. Lower, and R. Kurth. The viruses in all of us: Characteristics and biological significance of human endogenous retrovirus sequences. *PNAS*, 93:5177–5184, 1996.

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

[25] S. Rasmussen, C. Knudsen, R. Feldberg, and M. Hindsholm. The coreworld: Emergence and evolution of cooperative structures in a computational chemistry. *Physica D*, 42:111–134, 1990.

[26] T. S. Ray. An Approach to the Synthesis of Life. In *Artificial Life II*, pages 371–408. Addison-Wesley, 1992.

[27] S. Stepney, R. E. Smith, J. Timmis, A. M. Tyrrell, M. J. Neal, and A. N. W. Hone. Conceptual frameworks for artificial immune systems. *IJUC*, 1(3):315–338, 2005.

[28] P. Suber. *The Paradox of Self-Amendment: a study of law, logic, omnipotence, and change*. Peter Lang, 1990.

[29] S. Thrun and L. Y. Pratt, editors. *Learning to Learn*. Kluwer, 1997.

[30] C. Woese. *The Genetic Code*. Harper & Row, 1968.