Bio-Reflective Architectures for Evolutionary Innovation

Simon Hickinbotham and Susan Stepney

Department of Computer Science, University of York, UK
York Centre for Complex Systems Analysis
email: sjh518@york.ac.uk

Abstract

Computational reflection uses software architectures that are capable of self-modification at runtime. These systems have implementations between two extremes: procedural reflection, in which unlimited self-modification is available at the expense of infinite recursion; and declarative reflection, which uses pre-defined metrics to drive the self-modification and is hence limited in scope. Biological processes also exploit the concept of reflection, where natural selection drives the process of modification. The concept of a 'program' in computing has an analogy with an individual member of a species. The process of life is discretised into a series of autonomous systems, each of which creates modified versions of itself as offspring. This paper unifies the concept of computational reflection with biological systems via a new analysis of von Neumann's Universal Constructor. The result is a bio-reflective architecture that is capable of unconstrained self-modification without the problems of infinite recursion that exist in the computational counterparts. The new architecture is a blueprint for applications in Artificial Life studies, Evolutionary Algorithms, and Artificial Intelligence.

Introduction

In this paper we unify certain concepts from computational reflection (Maes, 1987; Smith, 1984) and Artificial Life (ALife). These concepts address how self-representation, autonomy and evolution contribute to 'living' systems. Each of these topics has aspects that are represented in the idea of *reflection* – computing which is 'about itself' – and the manner in which the genotype simultaneously specifies and is maintained by the phenotype.

As we describe below, computational reflection and biological systems have many things in common. A model of computational reflection gives new insight into biological systems. In addition, biological systems give a new perspective on the nature of computational reflection.

We introduce this topic with a summary what reflection means in computer science, and then go on to discuss the implications in ALife.

Computation without reflection: First we present a highly simplified model of *Conventional Computing* (CCOMP), so that the concepts we present below have a clear conceptual base.

In CCOMP, computers run *programs* that process *data*. On execution, both the program and the data are held as binary digits in RAM. The CPU 'reads' the program, which 'acts upon' the data. For our purposes, we can consider that the CPU executes one instruction at a time, and that instruction works on one data word. This is possible because the sequence of operations is specified by the program, and the data is organised into a set of related structures in RAM. If the program is written correctly, it processes the data in the manner intended, even though the CPU never 'sees' the entire program or data at any one time. Although there is no physical distinction between the program and its data, it is usual for the two to be treated separately. The data is *processed* by the program, meaning that some if it is changed or manipulated to form the output; the program is fixed.

The number of instructions needed to do anything useful to data is usually very large. In order to make it easier to write useful programs *high level languages* have been developed, which group sets of instructions together into useful commands. In this way, modern programming languages make it possible to write programs without intimate knowledge of the hardware that the programs run on.

We illustrate this concept in figure 1. This shows the relationship between the code base, the interpreter, and the code that is executing. The *code base* is the program written in some language. It becomes *executing code* via the action of the interpreter. (By *interpreter* we mean whatever process is accessing the code base and executing it.)

Although the von Neumann architecture that forms the basis of CCOMP has program-data equivalence at the word level, there is not usually a direct way for the executing code to reflect aspects of its computation back to the code base or the interpreter. Although some well known languages such as Java support such reflection, there is no requirement to use reflection when writing progams. Such feedback is needed if the program is to be verified, maintained and improved. In the absence of automated feedback mechanisms, these tasks are carried out by human programmers. Models

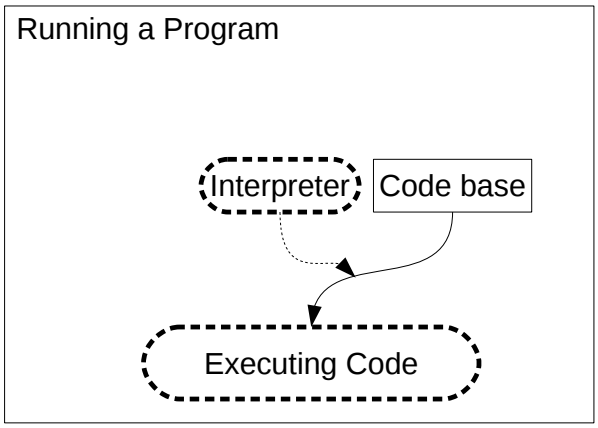

Figure 1: Running a computer program without reflection. The code base is analysed by the interpreter, and run as executing code. Solid boxes are data, Dashed boxes are running programs. Solid arrows indicate the provision of data. Dashed arrows indicate an action upon a process.

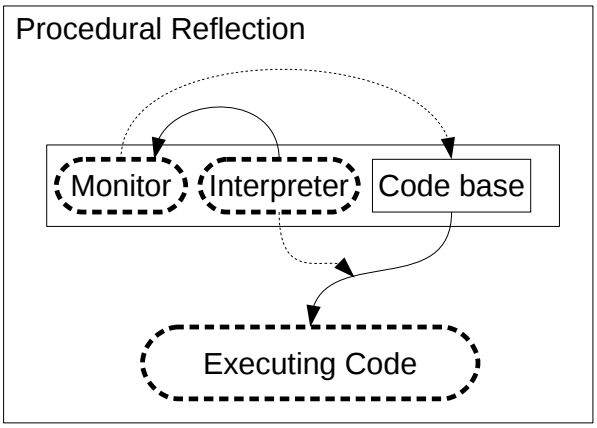

Figure 2: Procedural reflection. Key as in figure 1.

of computational reflection attempt to provide this feedback at runtime, which is guaranteed to provide the current context of the computation as it is being performed. Here we begin to see the relationship of reflection to ALife: the current context is the environment in which ALife systems must survive, and the process of life is the equivalent of the computational concept of runtime.

Next, we discuss reflection in abstract terms, and relate it to a series of related concepts in ALife. Then we review the way CCOMP has used reflection in different programming paradigms. The aim is to gather a set of observations on how reflection might work in ALife, which we present in the fourth section. We end with a discussion, and some proposals on ways to implement reflective ALife systems.

Computational reflection

"A reflective system contains structures which represent aspects of itself" (Maes, 1987). Reflection is any act of computing that is 'about itself': it is computation *about* the computation that is being performed, without direct reference to the *goal* of the computation. It is *self-inspection at runtime*, and a candidate definition of what comprises a living system. Self-inspection is of no use unless it is possible to act upon the outcome of the inspection, so CCOMP reflection allows *self-modification*: the ability to create (reify) new sorts of first class objects. We link this feature to living systems in the latter half of this paper.

Reflection requires that a representation of the code is available as data at runtime; the reflective process uses this to deduce which aspects of the program have particular computational features. A data structure representing (a model of) the program itself is created during execution of the program, and is used to modify the execution of the program at run-time. A special kind of interpreter (a) gives the running system access to data representing the system; (b) establishes the *causal connection* between the 'executing code' (the running system) and the 'base code' (the *system representation*). Causal connection guarantees that modifications to the executing system are reflected in the code base. All reflective operations depend on the maintenance of the causal connection to ensure that the code base remains a faithful representation of the executing code. Different methods of reflection use different ways to present the program representation as data to the running code (Maes, 1987).

A reflective system can bring about modifications to itself because it is able to generate and analyse data about its own computation. It is able to detect an issue in the executing code and modify the code base. The reflective interpreter reinterprets the code base *during execution*. By endowing a computational process with the power to monitor the computation that is being performed, systems are (theoretically) more able tolerate faults, organise their processing, and even organise their code base in the light of changing conditions (Smith, 1984). How this is achieved depends upon the mode of reflection being carried out. Two sub-classes of computational reflection are described below. The first, *procedural reflection*, allows reasoning about computation by running a model of the interpreter on a model of the code whilst the code is executing. The second, *declarative reflection*, attempts to avoid the costs of procedural reflection by abstraction of the properties of the executing code.

Procedural reflection

Procedural reflection encapsulates the role of the interpreter *within* the executing program, and assigns extra duties to it (Maes, 1987). The components of a procedural reflective architecture are shown in figure 2. The components of standard computation from figure 1 are all present. An interpreter process carries out the execution of the program, but

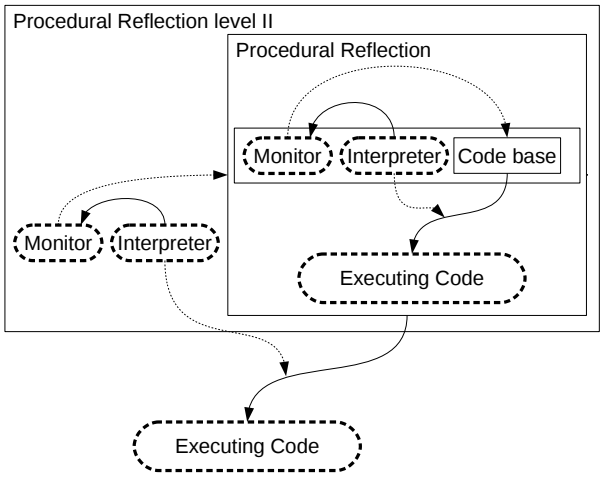

Figure 3: Meta-circularity of procedural reflection shown as two reflective layers. Key as in figure 1.

it also makes information about the computation available to a monitoring process, shown as *Monitor* in the figure. The encapsulated interpreter must guarantee a causal connection between the code base and the executing code. The monitor is able both to reason about the execution of the program, and to act upon this reasoning by making changes to the code base, so changing the execution of the program at run-time.

Procedural reflection requires that a (more or less) complete representation of the program is contained in the reflective layer. This means that it is possible to *generate* the layer below from the representation in the current layer using the interpreter. There is a clash here because the twin goals of *specifying* the system and being able to *reason* about it can be incompatible, leading to duplication of information at the very least.

Challenges in procedural reflection: The method of self-representation offered by procedural reflection has its challenges, centering on the problem of exactly *when* to spawn a process that is 'about' another process, since each process has a computational cost in terms of RAM and CPU. We enter the domain of *meta-circularity* when we realise that the reflective layer in figure 2 is itself a running program. If the interpreter has to have a complete representation of the relationship between the code base and the running program, it follows that a fully reflective architecture would need a second reflective layer, figure 3. Since the reflective representation is part of the running code, it must *also* be monitored, at a higher level. Following this reasoning, we see that the recursion in this model can extend *ad infinitum*, whereby a hierarchy of processes are spawned, each monitoring the process below and with only the process at the bottom doing any actual work. This problem is avoided by bending the rules slightly, letting the interpreters represent only parts of

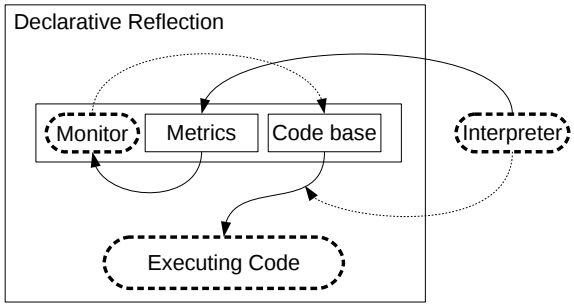

Figure 4: Declarative reflection. Key as in figure 1.

the system at each level and eventually deciding that further recursion is no longer fruitful.

In addition, the role of the interpreter in procedural reflection is complex since it has two tasks to perform: interpreting the code base, and feeding back information on the computation to the monitor. If we forsake the embedding of the interpreter within the reflective layer, some of these problems can be avoided.

Declarative reflection

Declarative reflection avoids the need to specify the code base exactly, and instead seeks to generate useful statements about the system (Maes, 1987), for example, information about the time and space complexity of the executing process. The interpreter sits outside the reflective process, and merely provides it with a set of metrics. The monitor can then act on these metrics and change the code base, which is then used as executing code by the interpreter (figure 4).

The advantage of this approach is that the danger of infinite recursion of reflective layers is greatly reduced (although still possible), and the duties of the interpreter are more clearly defined.

Challenges in declarative reflection The benefits of declarative reflection come at the expense of the ability of the reflective system to detect appropriate conditions that should be acted upon. The metrics can describe only *what* has been done by the system; it is much harder to give a description of *how* the effect has happened, making it more difficult for the Monitor to decide how to implement changes.

For this reason alone, reflective architectures are rarely purely declarative. Most reflective architectures use elements of both procedural reflection and of declarative reflection.

Reflective properties of ALife systems

We seek ways to apply reflective ideas from CCOMP directly to ALife systems, in the hope that the advantages of reflection can be emulated. However, reflection in biology is *different* from reflection in CCOMP. We are trying for a

'unified' treatment to reflective processes in biology, and so possibly find ways to improve reflection both in ALife systems and CCOMP generally.

Having reviewed procedural and declarative reflection, we must also describe an alternative approach to reflection based on phenomena and techniques observed in and inspired by biological systems. But first, we briefly review some of the issues in computational reflection from a biological perspective.

The absence of a 'designer': How can a biological system be 'about itself' in the absence of a pre-defined purpose? There are two parts to this question. First we consider how a design is specified, then we consider how this goal is met. (See also Dennett (1971) for a discussion of design stances.)

Biological systems use the genotype as a specifier of the phenotype via what Pattee (1982) calls the 'symbol-matter articulation', in which the specification of a machine (the symbol side) and its implementation (the matter side) are related to one another. This articulation is analogous to the requirement for causal connection in reflective languages, but it is at its strongest where the system exhibits *semantic closure* (see later).

Thus, in biology the design of the system seems to be absent from the model of reflection. But what is the goal of self-inspection and self-modification if there is no design to which the system can be compared? The answer is that biological systems introduce 'purpose' via evolution, by using populations of solutions and applying selection to them. Our goal in defining a bio-reflective architecture is to describe how these phenomena combine to yield a reflective system.

Rejecting declarative reflection: Declarative reflection offers a means of avoiding having a sophisticated interpreter in each reflective layer, and reduces the risk of infinite recursion. However, the declarative approach is rigid: it is difficult to detect when it is yielding insufficient information about the system, and it is difficult to implement new declarative statements when required.

Declarative reflection is like extrinsic fitness functions in Evolutionary Algorithms: it runs the risk of over-specifying the problem at hand and ignoring innovative solutions. The declarative approach is problematic in ALife because it adheres to an unchanging (and unchangeable) description of what the design is, implicit in the metrics that are used to monitor the system. It is difficult to define exactly what the declarative statements should be *a priori*, and so it becomes difficult to define what should be measured in order to detect what changes would be beneficial. The declarative approach embeds too much of the reflection in the 'physics' of the system (Hickinbotham et al., 2016), since the metrics are not under control of the 'biology', and so cannot be changed to improve its representation of the running system.

Rejecting procedural reflection: From the perspective of computer science, the two disadvantages of procedural reflection are that it places too many demands on the interpreter to allow an efficient implementation, and that the meta-circularity of the system leads potentially to infinite recursion.

The goal of reflection is to bring about automation in improvement of a computational system, in the same manner as natural selection in biological systems. Biological systems also exhibit recursion in that each organism is created by an earlier organism via a replication process. A key point is that biological systems are organised such that for most of its lifetime an organism is *autonomous*. In CCOMP, reflection requires the the ability to recursively spawn new reflective layers *ad infinitum*, but the 'recursion' in biology is the phylogeny of the individual. In bio-reflection, the individual does not need to spawn instances of its ancestors in order to monitor its state, since the relevant information is packaged up in its genome.

Reflectionless self-replicators Reflection involves holding a model of the code base and maintaining a causal connection between this and the actual execution of the model on the CPU. There is a direct analogy here with the relationship between the genotype and the phenotype in biology.

In ALife systems, models of biology are subject to experiments by computer simulation. Unlike biology, everything about these model systems is knowable, but everything (including all of the relevant physics) must be initialised, parameterised and implemented. There are also many assumptions about the appropriate representation of such simulations in mainstream computers. The attraction of this approach is that it makes clear the relationship between mechanisms of biological innovation and how these models of biology could be applied to (models of) computation.

Many self-replicating ALife systems exist, but these tend to be modelled on a hypothetical 'RNA world' in which each entity inspects an instance of itself in order to create a copy (Ray, 1991; Ofria and Wilke, 2004; Hickinbotham et al., 2010). These are *automata chemistries* (Dittrich et al., 2001): artificial agent-based systems in which each agent is a program.

Many ALife platforms contain instances of agents that iteratively manufacture copies of themselves. A mechanism for changing the copies, usually called *mutation*, is introduced in order to explore the design space of the system's universe. In this sense the program that the agents are running is *self-modifying*. This process of self-modification is central to mechanisms of reflection.

Although these systems have shown innovation, there seems to be an upper bound on the level of complexity they can attain, even though there is no theoretical limit on the innovation. Could this be related to reflection? We address this question by turning to the work of Von Neumann

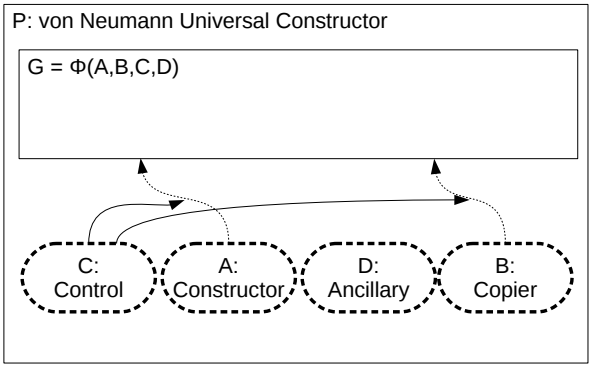

Figure 5: von Neumann's Universal Constructor Architecture. Key as in figure 1.

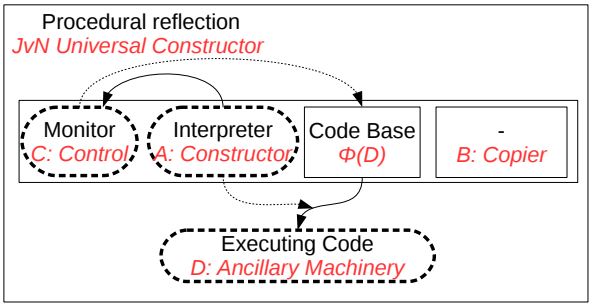

Figure 6: Comparison of components of procedural reflection (black text) from figure 2 and the Universal Constructor (red text). Key as in figure 1.

et al. (1966), in his *theory of self-reproducing automata*. The point, also made recently by the McMullin group in Dublin (Baugh, 2015; Hasegawa, 2015), is that these systems tend to reproduce by a process of *self-inspection*. von Neumann indicated that there are limitations to reproduction by this method, linked to the difficulty of 'reading' a machine of arbitrary complexity. We argue in addition that these systems are *reflectionless*; although they appear to be 'about' themselves, they are merely sophisticated *quines* (self-copying automata with no inputs) that make no reference to an abstract model of what they are. We have made similar points in (Hickinbotham et al., 2011). We emphasise here that although a system may have program-data equivalence, it is not guaranteed to be 'reflective'; to achieve this, further conditions must be met.

Universal constructors von Neumann's observations about the limitations of reproduction by self-inspection led to the development of his theory of self reproducing automata. He defined a set of sub-assemblies that together formed a Universal Constructor (UC). The original was cellular automaton-based, but the ideas translate to ALife and biological systems. We follow the notation of McMullin (2012) in the following.

The von Neumann architecture comprises four machines A, B, C, D plus their machine descriptions $\Phi(A, B, C, D)$, figure 5. The process of self-reproduction is divided into two parts, which allows machines of arbitrary complexity to be duplicated. Only one entity in the system is copied by inspection. This is G, which consists of an abstract description of everything else in the system: $G = \Phi(A, B, C, D)$. The remaining four machines function as follows. A is the *Constructor*, which can read G and construct (or express) functioning machines from their descriptions. B is the *Copier*, which can create copies of G by inspection (for this reason, G is usually a one-dimensional sequence of instructions). The operation of A and B with respect to G is governed by C, a *Control* structure. D is *Ancillary Machinery*, which carries out any other function of the system irrespective of the self-replicating assemblages just described.

We illustrate the overlap between von Neumann's Universal Constructor architecture and procedural reflection in figure 6. The layout of this figure follows the procedural reflection diagram in figure 2, and adds the UC nomenclature in red. All of the components of UC bar one are present in procedural reflection, but the naming conventions are different. What we have called the Monitor is called the Controller in the UC, but their roles are identical: to orchestrate the operations of the other sub-assemblies in the overall machine. The Interpreter is mirrored in the UC as the Constructor, which takes a description of a machine and creates the machine based on that description, in the same way that an Interpreter reads source code and creates a working manifestation of the code on a conventional computer architecture. The Code Base in the procedural architecture is represented in the UC as the symbol $\Phi(D)$. Both of these labels represent the abstract concept of a *description of a functioning machine*: D is the functioning machine, and Φ is a description operator. The executing code in the procedural reflection model is referred to as 'D: Ancillary machinery' in the UC nomenclature. The only component that UC adds to the procedural reflection model is the *Copier*, and a description of all the machines, not just the ancillary D, in G. The copier is responsible for duplicating the machine descriptions in Φ. We describe its role in bio-reflection below.

The layout of figure 6 is an unsatisfactory description of bio-reflection because it falls victim to the meta-circular architecture in the same way as shown as in figure 3, but the conceptual link is important for what follows. The UC nomenclature in this figure already gives some clues about what is missing from the model.

Bio-reflection

Having made some observations about reflection in ALife, we now propose a new architecture of self-modification, which we call *bio-reflection*, since its development from

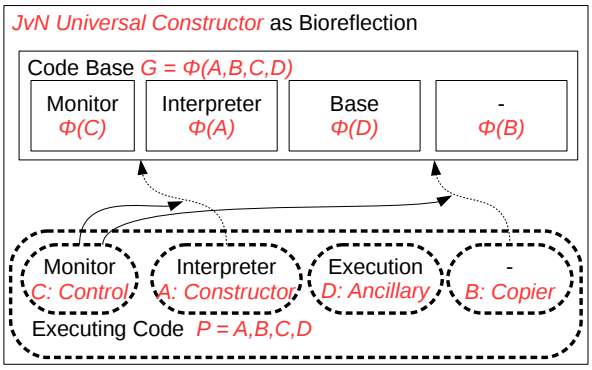

Figure 7: Bio-reflective Architecture. Universal Constructor terms are shown in red. Key as in figure 1.

CCOMP reflection is inspired by biological processes.

Inspection of the mapping between the UC architecture and the procedural reflection architecture in figure 6 shows that some of the components of UC are missing from it. Firstly there is no machine description for anything but the Ancillary Machinery in a single reflective layer. Secondly, there is no Copier.

The absence of descriptions of all the machine appears to be the feature that forces reflective systems into a recursive situation of figure 3. von Neumann solved this problem by specifying that *all* the components in the architecture be represented by abstract descriptions.

The copier is missing from the reflective architecture for two reasons. First, it is easy to copy source code in CCOMP. Second, the concept of reflection requires that the interpreter acts upon the source code *while the program is running*. From this perspective, why would one bother to have an extra machine in the architecture?

Why is a copier important? This is related to the absence of a designer. It is hard to say what a living organism is *about*, because there is no designer, and hence no *purpose* to the organism. This makes the concept of *control* rather more vague: what is it that the entity is being controlled *for*?

Due to mutation, variation in the expression of $\Phi(X)$ means that it is likely that a machine will fail sooner or later. Having a population of individuals insures against that. A population of machines is inevitably exposed to selection. By making the copier an intrinsic part of the machinery, we can ensure that successful individuals are reproduced in the population.

So bio-reflection works on *populations* of machines, which gives the control component of the architecture we seek. This is part of the way we avoid infinite meta-circular reflection. But we have to accept that populations have their own associated computational costs.

The bio-reflective architecture follows the UC architecture, but with some external considerations that we describe below. The overlap between the bio-reflective architecture and UC is shown in figure 7. There are two classes of entity: the *code base* and the *executing code* as in computational reflection. The difference is that *all* the executing machines in the system have a representation in the code base layer. The idea is that, following the UC architecture, the machine descriptions in the code base are sufficiently rich to allow the machines to be constructed from them, but sufficiently simple to be easy to copy by the copy machine. In this way, the system is self-contained (semantically closed), and there is no requirement for a recursive pattern of reflection to further organise the self-modification. Reflective actions that would have happened at a higher level of reflection are handled via changes to the representation in the code base (via the copier), and changes to the way that representation is interpreted by the constructor. These two components together guarantee causal connectedness in a semantically closed manner, without the need for any external specification (beyond the physics of the system, which should be minimised).

Facets of bio-reflection Mutation is different from self-modification, because of absence of a design: mutations merely modify, then *selection* identifies which of the modifications are improvements. Much of what a monitor has to do are moved to an external process that runs at the population level.

In order to generate the running code, several things must be brought to bear on the description of the machine. The functionality of the Interpreter is the most relevant to this discussion, but this functionality depends on the 'physics' of the system, which is not described the genotype, but is implicitly referenced by it. In this way, the abstract description is incomplete, but consistent with the executing machines. This feature allows us to sidestep the recursion that exists in computational reflection.

The architecture allows new kinds of machines to be reified via two routes. The first route involves the inaccurate copying of the code base via the action of the copier, leading to mutation in one of the machines. The second is a special case of this: an inaccurate reification of the interpreter machine has the potential to change the way the entire code base is interpreted, leading to a change in the way *all* machines are reified.

In this way, the semantics of the code base are self-contained because the machine that interprets the code base is encoded in the code base. This feature guarantees causal connectedness, but also allows the meaning of a code base to be interpreted differently depending on the nature of the interpreting machine, in what Pattee calls *semantic closure*. ALife systems that reproduce via self inspection do not have this feature, and so cannot be said to be reflective.

Semantic Closure and Causal Connectedness Causal connection is a major component of semantic closure, but it says nothing about the semantics of the system and how they arise. In the bio-reflective architecture, semantic closure means that the semantics of the machine descriptions are *embodied* in the relationship between the executing machines and their descriptions. This feature of the system is most prominent in the encoding of the interpreter, which has to read a description of itself, and construct a copy of itself from that description. If, by mutation or other error during construction, the function of the interpreter changes, then the whole meaning of Φ is changed.

Examples of such a change of meaning are well known in biology (Foster, 2007). For example, the 'SOS response' to DNA damage in *E. coli* involves the expression of RNA polymerases that are more able to successfully copy damaged DNA, but with lower fidelity, thus increasing the mutation rate whilst the population is under stress. A bio-reflective ALife system has the potential to emulate such phenomena.

A different continuum Maes (1987) indicates that most computational reflective architectures were somewhere between the Procedural and Declarative extremes. In most reflective systems, the code base is represented sufficiently explicitly to allow the reflection to occur, and the 'missing' parts of the code base are represented by declarative statements. The bio-reflective approach offers a different perspective on this: in biological systems the code base (genome) is augmented by the physical and chemical processes in the cell, allowing the enzyme 'machines' to be constructed by the ribosome; in computational ALife systems, the virtual physico-chemical processes are necessarily less complex, consisting of the function of opcodes, and the 'given' computational machinery (registers, stacks, etc) assigned to each individual.

Another continuum is to do with the relationship between mutation and selection: mutation pushes newly created automata towards a random/disordered state, but selection ensures that the reflective processes of interpretation and monitoring are maintained.

Discussion

Reflection and the design stance If computation is to be about itself, we need some definition of what the 'self' is. Here, biology is more straightforward, because the design of the organism is self-contained. The issue is more complicated in software because it is engineered: it has a designer, a purpose, and an implementation in source code.

We have an immediate difference between engineered reflection and emergent reflection. Engineered reflection requires the conscious act of building a reflective system. Computation can happen without reflection *precisely because* the computation is engineered: it has a designer. One of the problems is that reflection within these architectures holds the designer as a 'third option' in which the ultimate design can reside. The AI community have reflection to be about debugging, optimising etc, which is the origin of Smith's 'computing about itself' metaphor. *Reflective acts* are anything that is done 'about' the computation, such as debugging, optimising etc. This assumes the design is 'known' and that we are attempting to refine the implementation to reflect the design.

How can biological processes be reflective if they are not designed? This is a core question that we must answer if we are to build a bridge between computer and biological sciences. Firstly, we note that in some ways, the absence of a designer makes reflection simpler: the 'ultimate' design is simply the system itself; whether the design resides in the genotype or the phenotype is immaterial. Whereas in CCOMP, programs that did things could exist before reflective processes were available, in biology *this was not possible*: how could a process emerge that was 'doing' something before the system became self-referential? Without the self-referential process, the biology is nothing more than complex (carbon) chemistry. Only when a specification and an interpreter became available did life truly emerge.

We are not refuting RNA world with this argument; we are merely stating that even there, some RNA would be template, and some would be machine.

On the 'self' in Computer Science Unlike biological organisms, computer programs are *designed*. Following from this, Smith (1984) noted that we have an issue: what *is* the design of the program? Is it the concepts in the programming team's heads? The source code? The executing code? The answer appears to be a combination of all three: the concepts give the broad thrust of what needs to be done; the source code is an instantiation of these ideas, plus bugs; the executing code is what actually happens, which is what the source code is trying to persuade the interpreter to do. But some of the actions of the interpreter then become part of the design, and these are not necessarily in the source code.

McMullin's lab has attempted to build instances of the von Neumann replicating architecture in Tierra and Avida. Both implementations of the von Neumann architecture in these systems tend to collapse to their original RNA world configurations of replication, unless strict constraints are placed on the evolution. For example in (Baugh, 2015), the system could only be made viable by deleting any offspring with a different length from the parent. These results are important, because they allow us to identify features of automata chemistries that foster the more sophisticated self-reproducing entities described by von Neumann.

The model we propose could be implemented as an emulation of biology by allowing the monitor to act as a gene regulator only, or towards CCOMP-style learning system, by configuring the Monitor to pass 'message sends' to the

selection process. In either configuration, the reflective acts are fully autonomous and internally consistent.

Conclusion

The dream of AI is to have systems that adjust themselves to meet our needs, be they robotic, informational or biological. The foundation of these systems is that they are reflective: they are able to reason about themselves. This problem is encountered in ALife, Artificial Intelligence and CCOMP, as noted by Pattee (1982). By considering ALife as reflective systems, we are more able to draw from this wider body of research and move the field forward.

It is remarkable that the Universal Constructor design from 1949 is still relevant today. By casting it as a reflective system, new emphasis can be placed on the components of an ALife system, suggesting new avenues for research in both Artificial Life and Computer Science. To quote a review of the previous draft of this paper: "Interesting questions present themselves: should we be using reflective languages to build reflective replicators? Would self-awareness in a replicator, i.e., introspection into its own method of replication/ecological niche, enable an enhanced form of autoconstructive evolution a la Spector and Robinson (2002)? A Lamarckian-Darwinism where organisms simulate their offspring in sandboxes and hack their own genomes accordingly?"

CCOMP reflection does not account for copying of code bases in the reflective act, and the von Neumann architecture does not make clear the role of the monitor, or take natural selection into account within the model. With our bio-reflective architecture presented here, we have arrived at a consistent representation that clarifies the distinction between CCOMP and biology, and provides an intellectual basis for future ALife implementations.

Acknowledgements

This work was funded by the EU FP7 project EvoEvo, grant number 610427. We thank the anonymous reviewers for their perceptive comments.

References

Baugh, D. (2015). *Implementing von Neumann's architecture for machine self reproduction within the Tierra artificial life platform to investigate evolvable genotype-phenotype mappings*. PhD thesis, Dublin City University.

Dennett, D. C. (1971). Intentional systems. *The Journal of Philosophy*, 68(4):87–106.

Dittrich, P., Ziegler, J., and Banzhaf, W. (2001). Artificial chemistries – a review. *Artificial Life*, 7(3):225–275.

Foster, P. L. (2007). Stress-induced mutagenesis in bacteria. *Critical reviews in biochemistry and molecular biology*, 42(5):373–397.

Hasegawa, T. (2015). *On the evolution of genotype-phenotype mapping: exploring viability in the Avida artificial life system*. PhD thesis, Dublin City University.

Hickinbotham, S., Clark, E., Nellis, A., Stepney, S., Clarke, T., and Young, P. (2016). Maximising the adjacent possible in automata chemistries. *Artificial Life*, 22(1):49–75.

Hickinbotham, S., Clark, E., Stepney, S., Clarke, T., Nellis, A., Pay, M., and Young, P. (2010). Diversity from a monoculture: Effects of mutation-on-copy in a string-based artificial chemistry. In *ALife XII*, pages 24–31. MIT Press.

Hickinbotham, S., Stepney, S., Nellis, A., Clarke, T., Clark, E., Pay, M., and Young, P. (2011). Embodied genomes and metaprogramming. In *ECAL 2011*, pages 334–341. Springer.

Maes, P. (1987). Concepts and experiments in computational reflection. *ACM Sigplan Notices*, 22(12):147–155.

McMullin, B. (2012). Architectures for self-reproduction: Abstractions, realisations and a research program. In *ALife XIII*, pages 83–90. MIT Press.

Ofria, C. and Wilke, C. O. (2004). Avida: A software platform for research in computational evolutionary biology. *Artificial Life*, 10(2):191–229.

Pattee, H. H. (1982). Cell psychology: an evolutionary approach to the symbol-matter problem. *Cognition and Brain Theory*, 5(4):325–341.

Ray, T. S. (1991). An approach to the synthesis of life. In C. Langton, C. Taylor, J. D. Farmer, S. Rasmussen, editor, *ALife II*, pages 371–408. Addison-Wesley.

Smith, B. C. (1984). Reflection and semantics in Lisp. In *Proc. 11th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 23–35. ACM.

Spector, L. and Robinson, A. (2002). Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40.

Von Neumann, J., Burks, A. W., et al. (1966). Theory of self-reproducing automata. *IEEE Transactions on Neural Networks*, 5(1):3–14.