EvoEvo Deliverable 4.4

Computational reflective run-time platform

Due date:	M36 (October 2016)
Person in charge:	Susan Stepney
Partner in charge:	University of York (UoY)
Workpackage:	WP4 (A computational EvoEvo framework)
Deliverable description:	Computational reflective run-time platform: A calibrated, tested and documented reflective implementation of the platform specification (D4.2), suitable for developing a reflective evoevo component of an open-ended application (D4.5). A report containing a description of the platform and the CoSMoS approach "Simulation Platform".

Revisions:

Revision no.	Revision description	Date	Person in charge
0.1	Initial release for comment	24/10/16	Simon Hickinbotham (UoY)
1.0	Release version	27/10/16	Susan Stepney (UoY)
1.1	Code repository locations added	9/11/16	Susan Stepney (UoY)

Table of Contents

1. Introduction

The core area of research in this workpackage has been to address the issue of reflection in evolving systems. The established models of computational reflection (Maes 1987; Smith 1984) implement reflection either by recursive self-monitoring, or by a simpler process of applying metrics to software performance. The former cannot be directly embedded in an evolving system, whereas the latter does not have a facility to adapt as evolution progresses.

Facing these challenges, we have looked at ways to take abstractions of the framework from workpackage 4.1 and adapt them for EvoEvo at code level in such a way that reflection could be incorporated into the model. The abstraction that we used is inspired by von Neumann's universal constructor architecture (UCA) (von Neumann & Burks 1966). But the UCA has no *direct* relationship between the abstraction and conventional models of reflection. We have developed a new *bio-reflective* architecture (Hickinbotham & Stepney 2016), which re-casts computational reflection into an evolutionary framework.

2. Design and implementation

Our bio-reflective architecture has been designed, and various experiments based on the prior stringmol AChem have been implemented, as described in the following sections. First we summarise the platform details as necessary to make this document self-contained. The reader is refered to the source literature for the full details.

2.1. Bio-reflective architecture

Our new bio-reflective architecture is described in full in (Hickinbotham & Stepney 2016). In summary, it is a synthesis of concepts from:

1. von Neumann's Universal Constructor Architecture (von Neumann & Burks 1966)
2. procedural computational reflection (Maes 1987)
3. evolutionary algorithms
4. open-ended novelty mechanisms (Banzhaf et al 2016)
5. the EvoMachina architecture of active "machines" and passive "structures"

von Neumann's UCA contains a "tape" that encodes the system, and expressions of that tape that act on each other, and on the tape. The key concept is replication by copying the tape, then expressing its description as "machines" (Andrews & Stepney 2015), rather than replication by self-inspection.

The procedural reflection architecture has components analogous to some of the "machines" in the UCA, but only has access to the code of one of these machines: the application, not the expressor, copier, or controller machines.

Our bio-reflective architecture merges these two architectures, identifying the relevant machines. The UCA is augmented with (reflective) mutation and control; procedural reflection is augmented with access to the entire code base; a genetic population of individuals provides the route to implementing the reflective modifications of an individual; the resulting *bio-reflective* architecture supports the requirements of open-ended novelty.

2.2. Stringmol AChem

Stringmol (Hickinbotham et al 2010b) is an existing automata chemistry (Dittrich et al 2001) that we have exploited as a computational language and execution environment for most of our reflective EvoEvo research experiments reported here.

In stringmol, the `molecules' are programs encoded as strings of *opcodes* (single symbols, each of which specifies a computational operation to be performed). A stringmol chemistry operates in an abstract container, in which multiple pairs of molecular strings interact with each other. A (typically) aspatial physics engine gives pairs of molecules the opportunity to *bind* using a `soft' matching algorithm, where less good matches have a lower probability of binding. After binding, the molecular program executes, using both strings as program and data as determined by the sequence of opcodes.

An energy flux places an effective carrying capacity on the number of molecules in the system (although this depends on the size and properties of the molecules). Each opcode consumes a unit of energy when executed. A stochastic decay function removes molecules from the system with a fixed probability. Species of molecules must somehow be reproduced or they will disappear from the container. It is possible for the container to `die' when a mutation destroys a self-maintaining cohort of molecules and no molecules remain in the container.

Mutation can happen when a molecular program executes the copy opcode. A small chance of error is built into this operator, with the effect that a symbol is mis-copied (deleted, substituted, or inserted).

Molecular strings with certain computational properties are designed using the stringmol language. An experiment starts by seeding a container with such designed molecules, then running it under mutation. Different initial seed molecules result in different evolutionary behaviours.

2.3. Replicase stringmol

Stringmol has been used to explore replicase systems, by designing a seed molecule that can copy the molecule it binds to. The experiments seed a container with these replicase molecules. See (Hickinbotham et al 2010a) for details of its evolutionary behaviours.

Some of the experiments reported on below use the stringmol replicase molecule design to investigate EvoEvo properties in different contexts.

2.4. Bio-reflective stringmol

We have implemented the bio-reflective architecture in the stringmol language (Clark et al, submitted). This involved designing a new molecule, analogous to a genome that *encodes* expresor, copier, and application molecules. The initial container is seeded with genome and expressor molecules. When a Copier molecule binds to a Genome molecule, it executes and creates a (possibly mutated) copy of the genome. When an expressor molecule binds to a genome molecule, it expresses (possibly with error) the various molecules encoded on the genome (see figure 1).

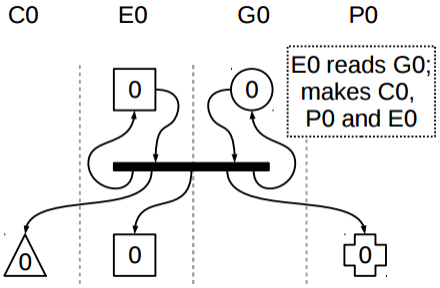

Figure 1. Normal operation of the Expressor in stringmol-UCA. Each shape represents a single molecule. Each molecular type has its own shape: Copiers (C) are triangles; Expressors (E0) are squares; Genomes (G) are circles; Payloads (P) are crosses. Each molecular species has its own column. The black bar represents a reacting pair of molecules. New molecules produced during a reaction are shown below the bar. Arrows show change of state. The Expressor binds to the Genome. The Genome *encodes descriptions* of the Copier, the Expressor, and Payload. The Expressor *interprets* the Genome to create new instances of these machines (possibly with error).

Some of the experiments reported on below use the stringmol genome/expressor molecule design to investigate EvoEvo properties.

3. Mechanisms for adding and modifying existing code

In an evolving system, the principal way in which code can be added or modified is through random mutation. However, a reflective architecture should go beyond this, and recognise when a situation *demands* change. This is seen in biological systems, where microbes under environmental stress increase their mutation rates in order to seek new adaptations (Foster 2007).

These self-modified changes occur at the population level: new individuals are produced with new features, uniquely adapted to prevailing conditions.

Thus, the bio-reflective approach to adding and modifying code is to have mechanisms in place that control both the rate and the manner of mutation in the evolving system. This is the key difference between our EvoEvo-based approach and techniques from genetic programming (GP): the reflective architecture requires that the code for mutation is *part* of the individual.

We see these mutation mechanisms emerge in the stringmol architecture. The root of mutation is based on a simple copy error, whereby a new type of program fragment is created which differs from its ancestor by a single instruction. In certain circumstances, this change can trigger a cascade of reactions that move the system to a new state.

Our work in EvoEvo has discovered two new routes to these states: space, and bio-reflection.

3.1. Spatial stringmol

Adding a spatial component to the stringmol engine has led to more stable replicator-based systems. This work, exploiting WP4 concepts, was performed under WP3, and is discussed in deliverable 3.4.8.3. Addition of spatiality lead to a reduction in parasitism in the population of code fragments.

3.2. Bio-reflective stringmol

In order to compare our bio-reflective architecture with the original replicase-based configuration, a direct representation of the bio-reflective architecture has been implemented in stringmol (Clark et al, submitted). It is necessary to reduce the copy-error rate and decay rate, since the implementation uses molecules that are many times larger than the replicase molecules.

The dynamics of the resulting system are markedly different from the original replicase design: parasitism is much less prevalent and is never responsible for collapse of the system.

The bio-reflective implementation shows that mutation cascades can lead to different interpretations of the way that information is stored on a gene, depending on the state of the gene decoding architecture. During 500 trials, we observed several instances of change in the semantic closure between genotype and phenotype, in which changes to the expressor molecule lead to a *different interpretation* of the genotype *without any change in the genome itself*. See figure 2 for one example; further examples of more complex cascades are given in (Clark et al, submitted).

The *meaning* of the Genome (given by the Expressor) is *encoded* in the Genome, and can change without the Genome sequence changing. This is a strong example of EvoEvo in action.

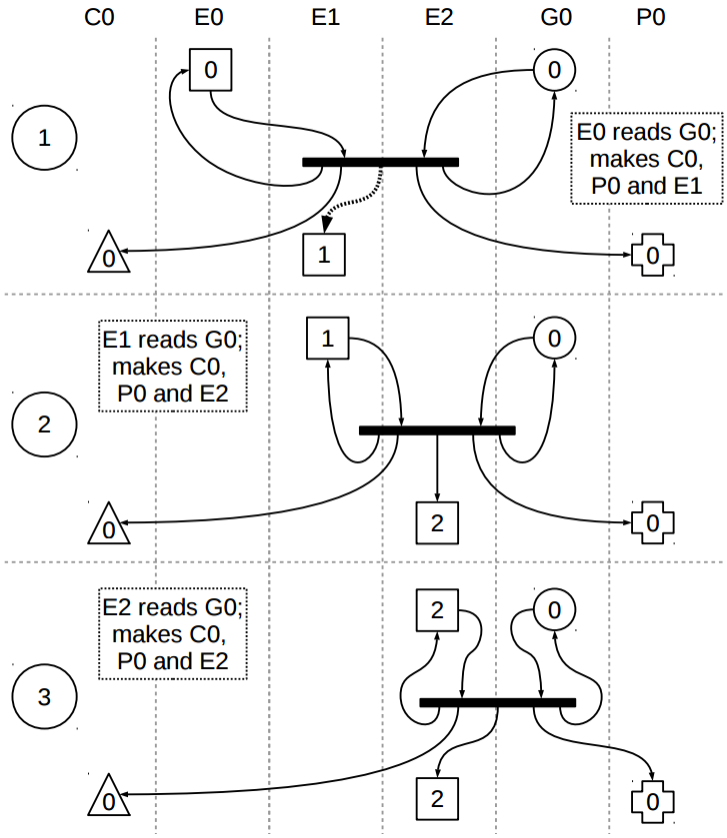

Figure 2. Semantic change without mutation of the genome. Key as in figure 4. (1) Mutation while expressing G0 leads to E1 instead of E0. (2) E1 interprets G0 differently from E0. Where E0 read the G0 as coding for E0, E1 reads the gene as coding for E2. Thus E1 expresses E2 *without error*. (3) E2 interprets G0 as coding for C0, P0 and E2.

4. A "Crash-Proofing" layer

One of the risks of evolving computer code is that the resulting execution profile brings the system down. This problem can be addressed by limiting the instruction set and placing restrictions on which data types can be passed to which operators. However, these approaches are rather restrictive for our needs.

4.1. Use of automata chemistries

Automata chemistries (Dittrich et al 2001) are ideal for experiments in this area, since they are designed to run code in a virtual sandpit without ever 'crashing the universe'. The challenge is to configure these chemistries such that innovative code structures are likely to evolve. This is achieved by:

- Permitting local interactions only: code fragments can only interact in pairs.
- Zero or minimal dependence and reference on a global control structure. Code fragments execute one instruction per global clock tick; code can only be executed if energy is available. Beyond these concepts of time and energy, there is no reference to global constraints.

By emphasising local interactions, arbitrary and unexpected modifications are most unlikely to have non-local effects. The majority of changes are deleterious, but these either fail to interact with local code fragments at all, or the interaction has no product. However, rare events can lead to changes in the *manner* of reproduction, even in Replicator-Parasite systems (Hickinbotham & Hogeweg 2016). When this is combined with a novel means of self-representation, our bio-refelective architecture (Hickinbotham & Stepney 2016; Clark et al, submitted), the result is a stable platform for evolutionary innovation.

4.2. Populations protecting against soft crashes

‘Hard’ crashes (where the system terminates via segfaults etc.) can be avoided by this strategy. ‘Soft’ crashes (where the system halts via squandered resources, or the complete loss of executable code, for example) cannot (be guaranteed to) be designed out of the system, due to subtle interactions between the environmental ‘physics’ and the individual ‘chemistry’. This is a problem of system design, both at the population level and at the code fragment level. Since the instruction set used in stringmol has proved sufficient to *generate* novelty, work in EvoEvo focussed on population-level considerations, which are more likely to *nuture* newly emerging beneficial traits.

4.3. Unintended biases in Automata Chemistries

One approach we have taken to this problem is to examine *selective biases* in existing automata chemistries, where features in the implementation had the effect of emphasising evolution towards parasitism (Hickinbotham & Stepney 2015b). We have identified several biases in Tierra, an early automata chemistry, which add unnecessary pressure to parasitism, and we have shown that by removing these biases the system is more able to resist the `race to the bottom' in terms of program length, increasing the likelihood of the emergence of innovative behaviours. This research has exposed the lack of theory regarding the design of stable automata chemistries (although recent work by Ackley and Ackley (2015) is breaking ground in this area).

5. Support for an interacting population of code fragments

5.1. Space

Our main work on this topic has been to experiment with the environment in which code fragments interact. Our prior formulation of the stringmol automata chemistry modelled interactions in a "well-mixed reactor": under the assumption that it was important for code fragments to have the opportunity to interact with any other fragment in the system. However, this approach could not deliver systems that were stable due to the emergence of 'parasitic' code fragments that hi-jacked the replicating component of the system. Due to the mixing of code fragments, these parasites were able to interact with the entire cohort of replicators resulting in extinction of the population. Moreover, because the only resource that was limited was the energy per time step, it was possible for the parasites to get themselves copied indefinitely.

Work conducted between York and Utrecht partners within EvoEvo has allowed models of Replicator-Parasite systems to inform the design (Hickinbotham & Hogeweg 2016). See deliverable 3.4.8.3 for further details. Before our work, it was thought that the addition of a spatial component to the AChem model was an unnecessary complication; we have shown that this is not the case: the spatial component is an important part of the design, eliminating the chance of 'soft' crashes via parasites.

5.2. Mass conservation

Parasites are a well-known problem in ALife, and can be lethal to the system by either destroying the self-replicating component, or squandering resources (Fortuna et al 2013; Ray 1991). One reason that parasites are so damaging is that there is no concept of material resource limitation in existing AChem systems.

We ran experiments in stringmol, using the replicase seed, to determine the effect of *mass conservation* on replicating systems (Hickinbotham & Stepney 2015a).

First, the system is given a fixed number of each type of opcode, with a uniform distribution, that can be used to make new molecules; decaying molecules release their opcodes to the pool. We found a "sweet spot" of opcode number (figure 3): too little material and the system cannot survive; too rich a source of material and the system has no incentive to explore other pathways.

Next, while still keeping the total number of opcodes constant at the "sweet spot" value, we searched for a distribution of opcodes that maximises evolutionary activity (figure 4), over 25 evolutionary runs. The high fitness results are much more evolutionarily active (QNN = 350-400) than the previous "uniform" opcode distributions (max QNN < 200, median QNN < 100). Interestingly, the copy operator '=' concentration is often (although not always) low in high fitness runs. The copy operator '=' has key functionality in the self-replicating system: how can low concentrations of this operator lead to self-maintaining runs with high evolutionary activity?

To address this question, we took the fittest of the 25 runs, and redid the run with the four lowest density opcodes (which includes '=') boosted by a factor of 10. We measured the QNN evolutionary activity over 20 runs of this boosted system. These all displayed only moderate evolutionary activity, whereas the unboosted systems have a much wider range of activities: being "on the edge" gives a higher risk of going extinct, but also a higher chance of more diversity.

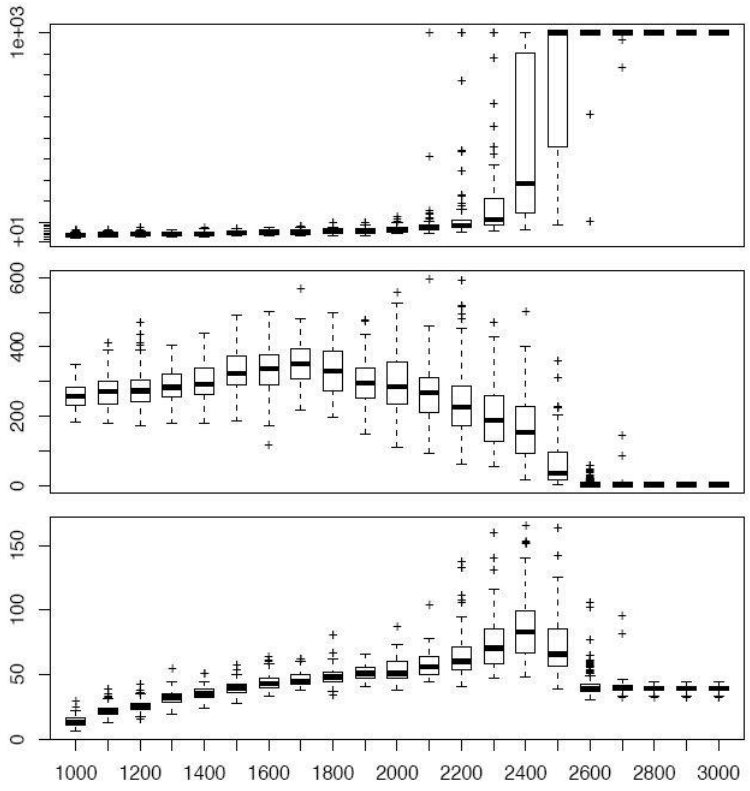

Figure 3. Existence of a "sweet spot". Lifetime (top), number of species (middle) and QNN evolutionary diversity (bottom) for a range of opcode concentrations 1000 < lambda < 3000. 100 runs at each concentration are evaluated. See (Hickinbotham & Stepney 2015a) for full experimental details

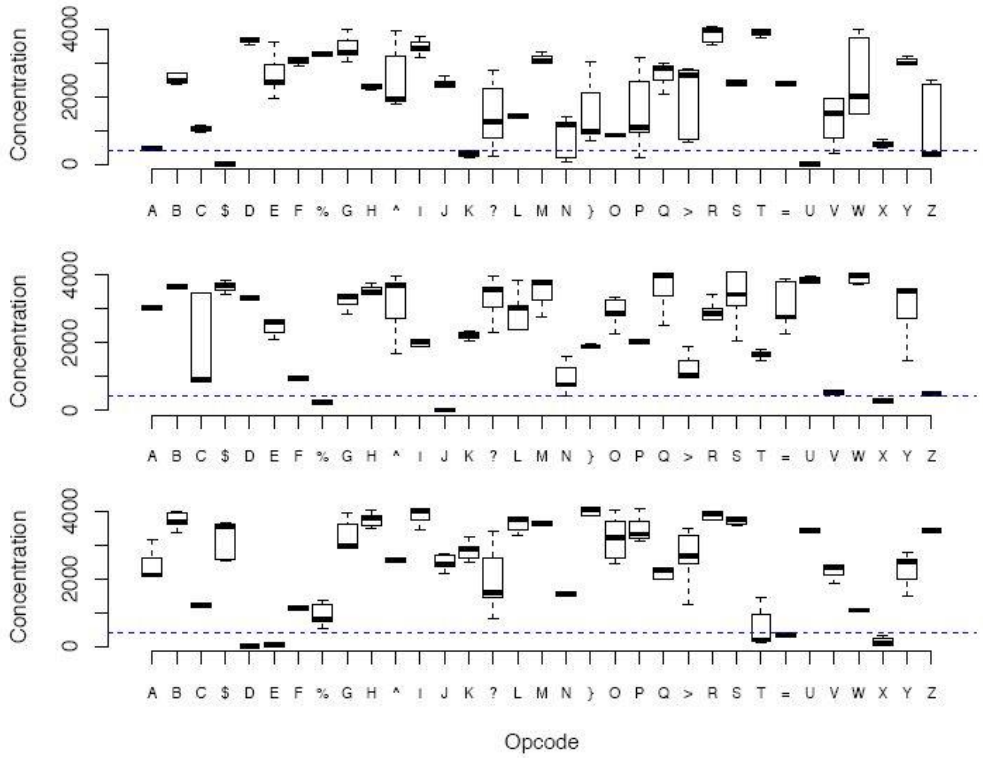

Figure 4: Evolved opcode concentrations for three typical GA runs (out of 25 runs in total). (top) a low fitness run, QNN < 100; (middle) a medium fitness run, QNN = 250; (bottom) a high fitness run, QNN = 350. The dashed blue line indicates a concentration of 400, which is 10% of the maximum concentration permitted in the GA. See (Hickinbotham & Stepney 2015a) for full experimental details.

These experiments demonstrate that mass conservation alone can push a system to new states *without external imposition of a mutation rate*. If an opcode is unavailable during replication, copying cannot occur; substitution or deletion are the only possibilities. The consequences of this simple observation are striking: under mass conservation laws, the 'rate' of mutation can change *automatically* if a system is stressed by resource limitation.

In addition, we found that it is possible to promote a more active search of program-space by limiting the opcodes that are directly involved in the self-replication program, because this forces replicators to seek innovative replication pathways.

6. Environmental interactions

Environmental interactions form the 'payload' component of the bio-reflective architecture. In this way, they can be made subject to reflective actions as much as the other components of the system that we have given more direct consideration to. Rather than attempt to develop a single language for all applications at this stage, we have looked at how Domain Specific Languages for specific, dynamic tasks might be evolved within this framework. This work is covered in deliverable 4.5.

7. Code locations

The stringmol code repository, including spatial stringmol, is at github.com/franticspider/stringmol

The specification of the Genome, Expresser, and Copier stringmols used to implement the bio-reflective architecture on the stringmol platform, are provided in Clark et al (submitted),

A stringmol web application, allowing experimentation with specific stringmol instances, is available at stringmol.york.ac.uk/webapp/

8. Conclusion

Our research has shown that, while it has not been possible to implement *conventional* models of computational reflection within an evolving system, a new synthesis of concepts in computational reflection with architectures for self-replication – our *bio-reflective architecture* – yields a more powerful model of reflection than has been available previously. These systems are most readily implemented in an automata chemistry. Our experiments show that these systems are capable of new levels of innovation, such as changes in the way a gene is interpreted.

AChem models of programming are a new area of research, and the formulation of the computational model is in its infancy. In addition to the theoretical contribution of the our *bio-reflective architecture*, we have made three important experimental contributions:

1. discovery of implementation factors that *bias* the system towards parasites
2. how *space* can protect a system from being overcome by parasites
3. how *mass conservation* can drive evolutionary pathways

Each of these considerations adds to the robustness of the system, allowing novel behaviours to emerge within the bio-reflective architecture framework.

9. References

References marked with (*) were produced wholly or in part by the EvoEvo project.

D.H. Ackley, E.S. Ackley. Artificial life programming in the robust-first attractor. In *ECAL 2015, York, UK, July 2015*, pages 554-561. MIT Press, 2015.

Paul S. Andrews, Susan Stepney. A Metamodel for the Evolution of Evolution. *ECAL 2015, York, UK, July 2015*, pages 621–628. MIT Press, 2015

(*) Wolfgang Banzhaf, Bert Baumgaertner, Guillaume Beslon, René Doursat, James A. Foster, Barry McMullin, Vinicius Veloso de Melo, Thomas Miconi, Lee Spector, Susan Stepney, Roger White. Defining and Simulating Open-Ended Novelty: Requirements, Guidelines, and Challenges. *Theory in Biosciences*, 135(3):131-161, 2016

(*) Edward B. Clark, Simon Hickinbotham, Susan Stepney. Semantic closure demonstrated by the evolution of universal constructor in stringmol. *submitted to Royal Society Interface*, 2016

Peter Dittrich, Jens Ziegler, Wolfgang Banzhaf. Artificial chemistries -- a review. *Artificial Life*, 7(3):225-275, June 2001.

Miguel A. Fortuna, Luis Zaman, Aaron P. Wagner, Charles Ofria. Evolving digital ecological networks. *PLoS Comput Biol*, 9(3):e1002928, 2013.

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

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

Simon Hickinbotham, Edward Clark, Susan Stepney, Tim Clarke, Adam Nellis, Mungo Pay, Peter Young. *Specification of the string-mol chemical programming language version 0.2*. Technical Report YCS-2010-458, Department of Computer Science, University of York, June 2010b.

(*) Simon Hickinbotham, Paulien Hogeweg. Evolution towards extinction in replicase models: inevitable unless.... In *2nd EvoEvo workshop, Amsterdam*, 2016.

(*) Simon Hickinbotham, Susan Stepney. Conservation of matter increases evolutionary activity. In *ECAL 2015, York, UK, July 2015*, pages 98-105. MIT Press, 2015a.

(*) Simon Hickinbotham, Susan Stepney. Environmental bias forces parasitism in Tierra. In *ECAL 2015, York, UK, July 2015*, pages 294-301. MIT Press, 2015b.

(*) Simon Hickinbotham, Susan Stepney. Bio-reflective architectures for evolutionary innovation. In *ALife 2016, Cancun, Mexico, July 2016*, pages 192-199. MIT Press, 2016.

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

Thomas S. Ray. An approach to the synthesis of life. In C. Langton, C. Taylor, J. D. Farmer, S. Rasmussen, editor, *Artificial Life II*, pages 371-408. Addison-Wesley, 1991.

Brian Cantwell Smith. Reflection and semantics in Lisp. In *Proc. 11th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 23-35. ACM, 1984.

John von Neumann, Arthur W. Burks. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.