Innovation, Variation, and Emergence in an Automata Chemistry

Susan Stepney[1,2], Simon Hickinbotham[1,3]

[1]York Cross-disciplinary Centre for Systems Analysis
[2]Department of Computer Science, University of York, YO10 5DD
[3]Department of Electronic Engineering, University of York, YO10 5DD
susan.stepney@york.ac.uk

Abstract

Open-ended novelty is one of the goals of ALife. We use a recent definition of open-endedness, stated in terms of system models and meta-models, to demonstrate how the Stringmol Automata Chemistry achieves variation, innovation and emergence in a replicator-parasite system. We also show how Stringmol's self-modifying code allows certain of these novelties to be exploited within the system itself, while others are only externally observed.

Introduction

Open-ended novelty is one of the goals of ALife (Bedau et al., 2000; Taylor et al., 2016), yet the definitions of what precisely is open-endedness, or even novelty, can often be vague and difficult to use to evaluate and compare systems. Banzhaf et al. (2016) provide a more concrete definition in terms of models and meta-models, which can be used to categorise three classes of novelty – variation, innovation, and emergence – and hence three levels of open-endedness.

Here we describe how the Automata Chemistry Stringmol (Hickinbotham et al., 2016) has demonstrated all three levels of novelty as defined by Banzhaf et al. (2016). The focus of this paper is on the classification of the novelty that we have observed arising in evolving Stringmol systems; the detailed Stringmol results are described elsewhere (Hickinbotham et al., 2016; Clark et al., 2017; Hickinbotham et al., 2020).

Models and meta-models

Modelling is a powerful tool in general. A model of a problem domain can be used: to clarify domain concepts, entities, relationships and behaviours; to specify what should be implemented; to capture and systematise what is observed. The model of the concepts used to build these models is called the meta-model, which can be used at a higher level to clarify, specify and systematise the concept in different modelling paradigms. See Stepney et al. (2018) for a discussion of modelling in the context of complex systems simulation.

Banzhaf et al. (2016) introduce a means of classifying open-ended novelty in systems, distinguishing three types

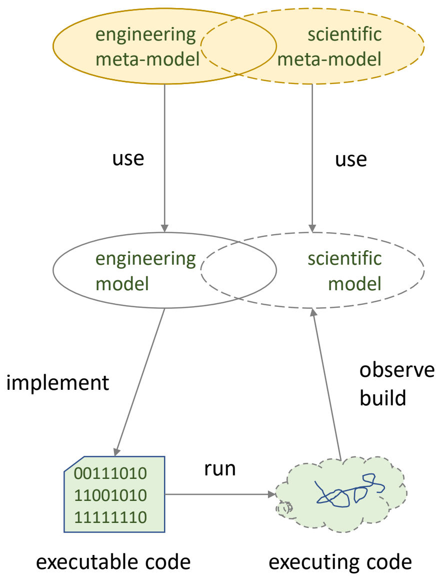

Figure 1: General framework for modelling novelty (adapted from Banzhaf et al. (2016)). The bottom layer is the system, as executable code that when run exhibits behaviours. Features of a design and its implementation are described in two overlapping domain *models*: the engineering model specifies the implementation design; the scientific model captures the observed system behaviours and concepts. The entities in the domain models are further described by a meta-model, which captures the domain concepts.

of novelty. The classification is relative to a particular model and meta-model of the system under investigation.

There are two distinct types of model (and associated meta-model) involved in Banzhaf et al. (2016)'s definition (figure 1). In typical system development, the *engineering model* is the specification used to implement, or engineer, the system under investigation; everything in the engineering model is instantiated in the implemented system; the implementation is a refinement of the engineering model. The *scientific model* is built from observations of, and deductions about, the system as its behaviour unfolds; everything in the scientific model is used to explain the observed system; the scientific model is an abstraction of the observed system. Entities that are both implemented and observed can exist in both models (although possibly in different forms); implemented but unobserved entities will exist only in the engineering model, while observed but not explicitly implemented entities (such as emergent properties) will exist only in the scientific model.

Types of open-ended novelty

Using these concepts of model and meta-model, Banzhaf et al. (2016) define three types of novelty that can occur in systems.

Type 0 novelty, or *variation*, is *novelty within the model*, such as the appearance of new instances or values of existing types. **Type 1 novelty**, or *innovation*, is a *novelty that changes the model*, such as speciation resulting in the introduction of a new class or type in the system model. **Type 2 novelty**, or *emergence*, is a *novelty that changes the meta-model*, such as a major transition (Maynard Smith and Szathmáry, 1995) resulting in the introduction of a new meta-class or type in the system meta-model.

Banzhaf et al. (2016) give examples of applying this definition, showing different kinds of novelty seen in Conway's Game of Life (Gardner, 1970; Berlekamp et al., 1982), in experimental evolution, in genetic algorithms, in genetic programming, and in the Tierra system (Ray, 1992).

Extrinsic and intrinsic novelty

Novelty is defined relative to a model or meta-model. We have two kinds of models: engineering and scientific, so we have correspondingly two types of novelties. Following the terminology of Taylor (2019), we call novelty identified in only the scientific model '*extrinsic* novelty', and we call novelty somehow internalised and reified by the system into its code, and hence in its engineering model, '*intrinsic* novelty'.

An example of an extrinsic novelty, observed only externally, occurs in a boids simulation (Reynolds, 1987). Flocking is an observed concept in the scientific model (which we build in our heads as we watch the simulation run), but does not explicitly appear in the engineering model (the specification and code of how the boids interact).

For such extrinsic novelties to be exploited and built upon by the system itself, they need to be somehow folded back into the engineering model, becoming *intrinsic*, changing the implementation, thereby available to support further novelties. This can be done externally, say by a programmer modifying the code. But for a truly open-ended system, this needs to be done internally, by the system itself (Stepney and Hoverd, 2011).

For example, in a Reynolds boids system, we would want to allow certain entities to interact with, relate to, or otherwise exploit, the emergent flocks. We would want the flocks to somehow become reified in the code, so that further concepts could emerge from their actions. That is, we would want the emergent to somehow become more, or other, than the mere aggregation of its components. In nature, this happens intrinsically: a bird might 'notice' a flock as a whole from a distance, rather than notice the individuals within it, as a consequence of the way its vision works on distant objects. In a different example, an emergent biological cell might start acting as an integrated whole because of the physical forces in its membrane making it cohesive. In a computational system, such new 'noticing' or 'cohesion' does not happen unless the simulation has code to make it so. To do so requires a change to the implementation code that adds the necessary extra 'physics' that happens 'for free' in a natural system; this change to the code happens in concert with a change to the engineering model.

Many current simulations exploring open-endedness are written in high-level languages that cannot change their code at run time; these can at best exhibit only extrinsic innovation or extrinsic emergence (figure 2). Although extrinsic novelty can occur, without self-modifying code the system cannot build on this emergence to produce further emergents.

Some high-level languages do allow code modification at run-time, and may be potential candidates for higher type novelties (Stepney and Hoverd, 2011). The approach we describe here to achieving mutable code structures is through self-modifying code in a low-level assembly language (figure 3).

Self-modifying code results in a rather different relationship between the engineering model and the implementation than is seen in typical software development. In top-down development, the model is built first, then the code derived from it. With self-modifying code, the implementation changes itself, and then we change engineering model (and possibly meta-model) to bring it into correspondence with the intrinsic code changes. These changed models need not capture what happened earlier in the system, if the code changes are radical enough.

Stringmol evolution

Automata Chemistries

Automata Chemistries (Dittrich et al., 2001) are a type of Artificial Chemistry where the 'molecules' are assembly language programs or program fragments. Some well-known AutChems are Tierra (Ray, 1992), Avida (Adami and Brown, 1994; Johnson and Wilke, 2004), and Amoeba World (Pargellis, 2001; Greenbaum and Pargellis, 2016). An analogous approach of an artificial chemistry whose molecules are constructed from low-level primitives is the object-oriented combinator chemistry (Williams, 2016).

Stringmol is an AutChem developed to explore RNA-world like (Hickinbotham et al., 2016) and DNA-world like

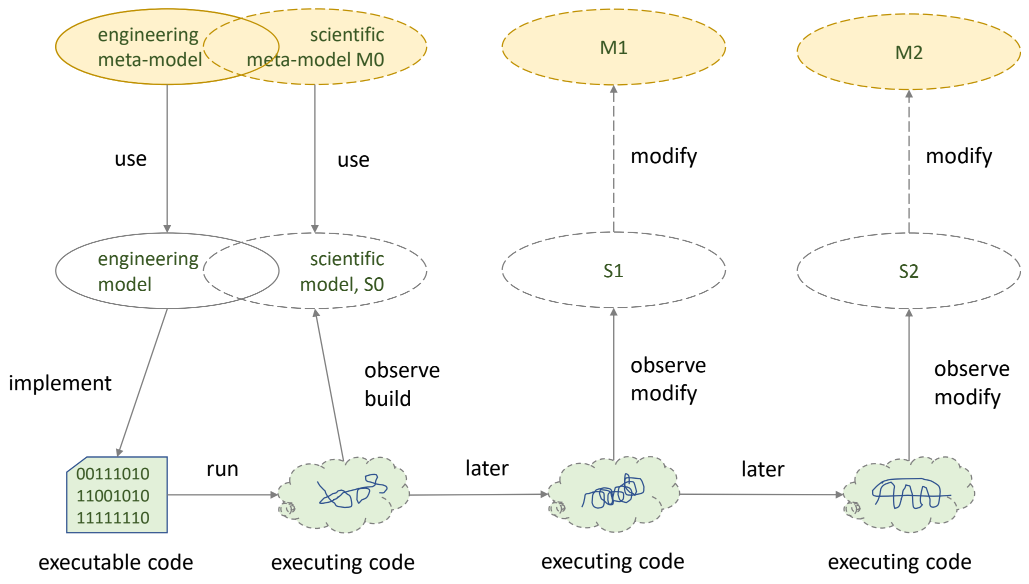

Figure 2: Classical simulations, without self-modifying code. The code is unchanged, but the behaviours on execution can show novelties. The scientific model is updated and modified to capture the observed novelties.

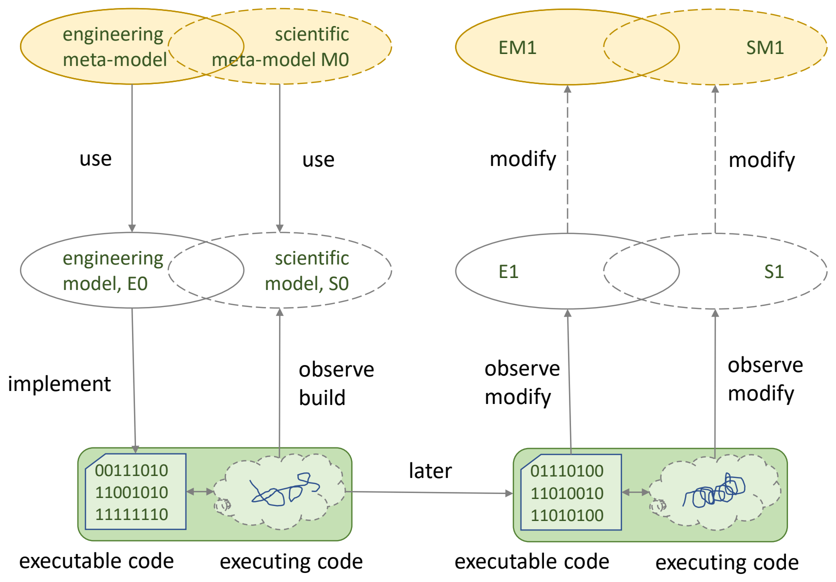

Figure 3: Self-modifying simulations, where execution changes the executable code. Behaviours on execution can show novelties due to code changes. The scientific model is updated and modified to capture the observed behavioural novelties, and the engineering model is updated and modified to capture the observed code novelties.

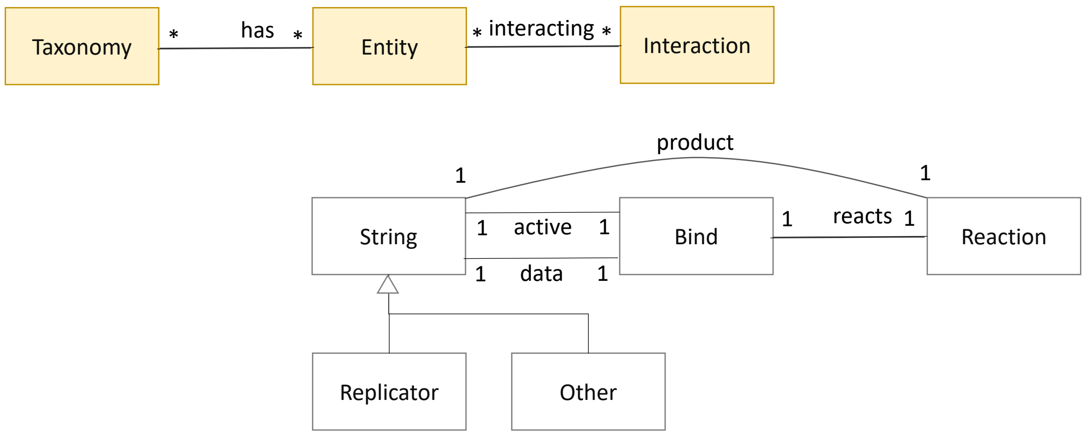

Figure 4: Initial meta-model (top, shaded) and model (bottom). Variation in the system is contained within these representations.

(Clark et al., 2017) interactions. Unlike many AutChems, in Stringmol, individual molecules ('stringmols') do not execute in isolation; rather, they bind, and one executes its program, using the other as data. For example, a 'replicator' is a stringmol A that binds to stringmol B, and executes its program to copy the data of B, thereby replicating stringmol B: A+B $\rightarrow$ A+B+B. This is the simplest case; the details become more complicated as mutation (incorrect copying) occurs. It is this extra complication that leads to the various types of observed novelty described below.

A Stringmol experiment typically comprises adding a collection of the hand-designed replicator or other specific string into a (well mixed) 'bucket' (Hickinbotham et al., 2016; Clark et al., 2017) or spatial arena (Hickinbotham et al., 2020). Strings react, and initially replicate as designed. However, a low rate of mutation on copy results in gradual evolution of the stringmols. There is no intrinsic fitness function, only the extrinsic fitness of survival. This gradual evolution results in novelties, in the scientific (observational) model, and also in the engineering model through code changes in the evolved strings.

Initial model and meta-model

The three types of novelty – variation, innovation, emergence – are defined with respect to a model and meta-model of the system. The initial model of Stringmol is sketched in figure 4. As is the case for many such systems, this model and meta-model were not made explicit in the original development, and are here reconstructed *post hoc*. Similar *post hoc* constructions can be done to analyse novelty in other systems, such as in the examples discussed in Banzhaf et al. (2016). We use UML as the modelling language here, but this should not be taken to imply that the system has an object-oriented implementation. We use the convention that concepts that occur in only the scientific model are indicated by italic text names and dashed boxes (used in figure 6): these are extrinsic concepts, not reified in the engineering model.

The initial meta-model captures the notion that we are designing a system comprising a collection of entities that interact to produce new entities, and that the types of these entities will change due to imperfections during the interactions. So it includes these three main (meta-)classes (figure 4, top):

- Entity: the concept of the entities in the model
- Interaction: the concept of interactions in the model; interactions involve entities
- Taxonomy: the concept of classification within the model; taxonomy applies to entities. The taxonomy is based on behavioural traits (for example, ability to replicate) not on structural features (specific code sequences)

The model is an instance of the meta-model that describes the 'virtual world', a collection of interacting and executing assembly language strings. We do not model the implementation of the underlying 'physics' that provides the reaction engine for these strings because these are immutable in the engineering model and we are unlikely to reify anything from the scientific model into this aspect of the implementation. There is no need here to provide a single complete model of the entire system; our goal is to clarify and define the aspects that are linked to novelty production.

The initial model has the following classes (figure 4, bottom):

- String (instance of meta-model concept Entity): instances are the individual stringmols in the system. Stringmols are made of a sequence of opcodes (assembly language instructions, not shown); the copy opcode executes with a small probability of error.
- String has two subclasses:
  - Replicator: instances are the hand-crafted stringmol that can bind to and copy instances of itself (which comprises the initial state of the system in our exper-

iments), and any mutations of this that arise as the system evolves that can still replicate
  - Other: instances that cannot replicate (which we expect to appear via mutation as the system evolves), they may have other as yet unrecognised behaviours

  The subclassing relationships (UML inheritance associations) are instances of the meta-model concept Taxonomy.
- Bind (instance of meta-model concept Interaction): instances are pairs of bound strings, one designated the active string, one designated the data string. The binding behaviour occurs with a probability (attribute, not shown), derived from the composition of the two strings.
- Reaction (instance of meta-model concept Interaction): instances are the executing bound pairs of strings, downstream from the starting point on the active string, defined by the bind location. A reaction produces a product string. If the active string is a replicator, the product is a (possibly mutated) copy of the data string. A reaction has a rate (attribute, not shown), the number of timesteps it takes to complete its execution.

Variation: changes within the model

The initial model and meta-model in figure 4 is both the engineering model and the scientific model (both implemented, and observed). The initial strings in the system are a population of identical instances of the hand-crafted Replicator, but the reactions are designed to be 'imperfect', producing mutant strings with low probability. Some of these mutants may still be replicators (variant strings, but with the same replicator behaviour), but some will have evolved out the replicator behaviour; they may have other behaviours. Our initial engineering model explicitly recognises this possibility: the design of the Stringmol language allows for both replicators and non-replicators to be implemented.

On execution of an initial system comprised purely of a well-mixed tank of replicator stringmols, we do indeed observe new instances of Replicator that differ in their detailed composition, but not in their replication behaviour; later we observe instances of Other, instances of strings that have mutated so that they not longer exhibit the replication behaviour. Also, these new instances have different bind probabilities, because of mutations in the binding region, and different reaction rates, because of different string lengths and different amount of looping behaviour on execution.

These new instances, probabilities, and rates are observed examples of variation, or type 0 novelty. No change to models or meta-models are needed to describe these novel stringmols or their behaviours.

Intrinsic Innovation: updates to the model

On running our system for longer, we see more novelty, and we start to be able to classify new classes of strings, and a different type of reaction.

Firstly, the Other subclass gets refined, by a combination of mutation (that introduces variation) and selection (that fixes certain variants in the population). These variants fall into two distinct subclasses. Parasites can be replicated by Replicator instances, but do not act as replicators when active; they lack the self-replicating code (Colizzi and Hogeweg, 2016; Takeuchi and Hogeweg, 2008). The instances of the now refined subclass Other cannot be replicated, and do not act as replicators when active.

Notice that these definitions are hinting that taxonomic properties such as 'parasite' may better be classed properties of Bind or Reaction, as they involve a definition of a stringmol's behaviour both when active and when the data: R+P $\rightarrow$ R+P+P, but P+R $\not\rightarrow$ P+R+R. However, this realisation is based on later knowledge, obtained from observing and analysing more complex kinds of reactions. At the time being described here, individual stringmols were identifiable and classified as replicator, parasite, or other, because the situation had not evolved sufficient distinctions to make a more specific classification necessary, or even expected.

Secondly, we see instances of reactions that occur, but produce no product string. The initial replicator string works by binding to string B, executing a program that adds a (potentially slightly mutated) copy of B to the end of string B, then cleaves off the copy, leaving the original string B plus the product copy of B: A+B $\rightarrow$ A+B+B. There are several ways a 'no product' variant of this occurs. First, the strings bind and execute, but no change occurs: a null operation: A+B $\rightarrow$ A+B. Second, the strings bind and execute, and the execution changes (overwrites) B without making a copy: A+B $\rightarrow$ A+B$'$. Third, a copy is added to the end of B, but no cleave occurs, leaving a doubling of the original B but no third product string: A+B $\rightarrow$ A+BB. The second and third variants can combine, producing a changed B with a (mutant) copy attached: A+B $\rightarrow$ A+B$'$B. We also see other reactions that produce multiple product strings: A+B $\rightarrow$ A+B+C+D. Such variations require a change to the cardinality of the String end of the product association in the model.

Thus we need to *change the model* to encompass these different two novelties (figure 5), so we have examples of type 1 novelty, or innovation. If this change were merely an external observation, only the scientific model would need to be updated, and we would have extrinsic innovation. However, the code has been changed by mutation, so there is also a change to the engineering model, and hence we have cases of intrinsic innovation.

Extrinsic emergence: updates to the meta-model

On running our system for even longer, we observe more complex reactions beginning to occur, which requires a rethink of our original taxonomy.

For example, we have identified *hypercycles* in earlier Stringmol experiments (Hickinbotham et al., 2016): pairs of

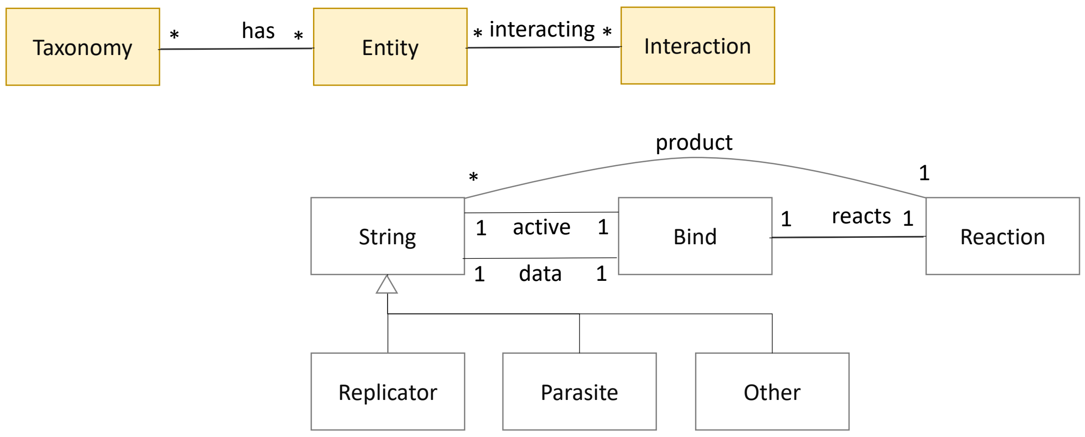

Figure 5: Innovation: unchanged meta-model (top, shaded); modified model (bottom): new subclasses, and changed product association cardinality.

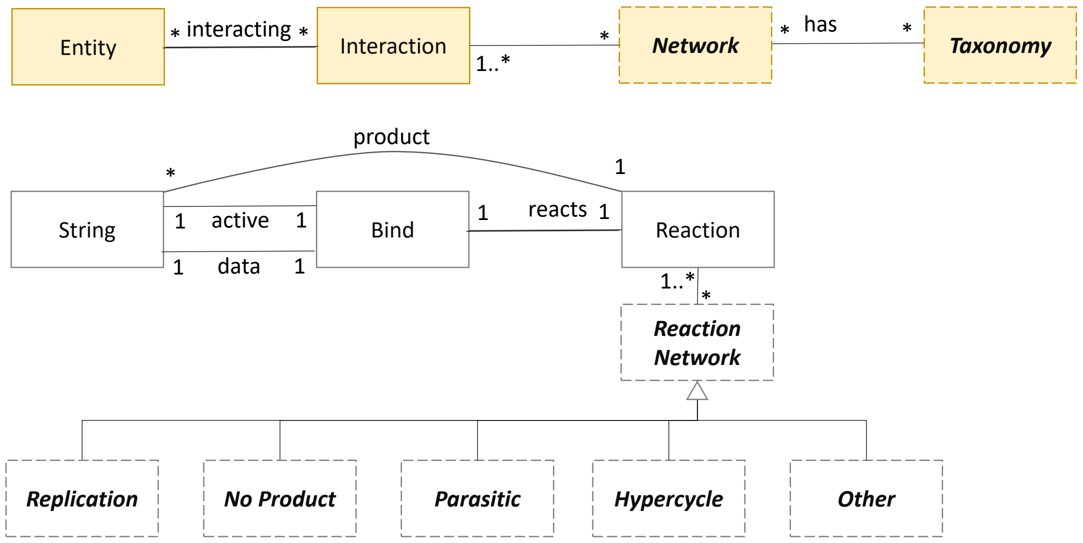

Figure 6: Emergence: modified meta-model (top, shaded) and modified model (bottom).

strings that can mutually replicate, but cannot self-replicate. Digging deeper into the analysis here, it becomes clear that *networks* of reactions are happening: properties cover the behaviours of multiple reactions; for example, being a hypercycle requires two reactions (A can copy B: A+B $\rightarrow$ A+B+B, and B can copy A: B+A $\rightarrow$ B+A+A), and also requires two counterfactual reactions (neither A nor B are able to copy instances of themselves: A+A $\not\rightarrow$ A+A+A, B+B $\not\rightarrow$ B+B+B).

Additionally, the case of parasites has grown more complex, as more variant stringmols evolve and fix in the population. It is no longer simply the case that replicators copy parasites but parasites do not copy replicators. Instead, some replicators copy only some parasites, and some parasites do not copy only some replicators. Indeed, even the property of being a replicator depends on the partner string: some partners are copied, others are not. Parasitism and replication are now understood to be properties of reactions, or sets of reactions, rather than of individual stringmols.

So at this point it is no longer possible to identify properties of reactions in isolation; rather we must consider the role a reaction is playing in the immediate community of stringmol variants and other reactions. These identifications are necessary to explain the dynamics of the system: without identifying these features, we cannot explain the competitive advantage that these properties confer.

Note that in automata chemistries that reproduce by binary fission in the execution of a single molecular program, such as Tierra and Avida, parasites are type 1 novelty (innovation), because (an appropriate variant of) figure 5 explains them adequately: there is no need to consider a network of reactions to identify their properties because a parasite there is definable in isolation, as having a lack of self-replicating code. Reactions in those systems do not involve two molecules interacting in the way the stringmols do: Tierran parasites execute code for copying that exists elsewhere in the system; Avidan parasites attempt to 'inject' code into non-empty cells and 'steal' CPU cycles.

To accommodate these new observations of stringmol behaviours, we need to update the meta-model to have the concept of a Network of multiple interactions, and a Taxonomy of these Networks, rather than of Entities (figure 6): it is no longer appropriate to say that a particular stringmol is a replicator, rather we now say that a particular reaction is a replication reaction. We can then update the model to include the class Reaction Network, an instance of the meta-model concept Network, with these reaction networks subclassed into ones with various properties, the subclassing (inheritance association) being instances of the new Taxonomy meta-model class.

Since we need to *change the meta-model* to encompass these novelties, we have examples of type 2 novelty, or emergence. This is a change only to the scientific model; Reaction-Network is an abstraction of observed sets of reactions; the Taxonomy subclassing instances are our model of what is happening; the system itself has not engineered these concepts into its code directly. Hence we classify this as *extrinsic emergence*.

Banzhaf et al. (2016) identify one particular form of emergence as a *major transition*. This special kind of emergence occurs when the novelty introduces a new *level of structure* of the concepts in the meta-model: where the meta-model includes concept X, a major transition is defined as adding the concept of an aggregate or system of Xs. Here, the added class Network in the meta-model is such an addition: it is an aggregate or system of interactions. So this particular *extrinsic emergence* is an *extrinsic major transition*.

Discussion and Conclusions

We have shown how type 0, 1, and 2 novelties can be demonstrated in the Stringmol system. Intrinsic novelties can occur in the Stringmol system because it supports self-modifying (assembly language) code.

We have shown how such novelties are relative to a particular model and meta-model: a different model could result in a different classification of these novelties. This should not be surprising: to say something is novel, or different, we have to say different *from what.*

If we had started our experiments with the model in figure 6 then we would not be able to claim these features as type 1 or 2 novelty for our system. Here we have been parsimonious with our models, including only the components needed to explain the code and observations at any one time. However, even if we start from a model that already contains some predicted innovations and emergents, in an open-ended system there will be others discovered later, as the system evolves further. Indeed, this is a necessary property for open-endedness, for the continual production of innovation or emergence (Banzhaf et al., 2016). The system eventually will have to move outside its model (to create yet more innovation) or its meta-model (to create yet more emergence), no matter how large or prescient the original models were. This captures the essence of open-endedness: that the innovations and emergents are ever new kinds of novelty, not simply a repetition of previously encountered novelties.

References

Adami, C. and Brown, C. T. (1994). Evolutionary Learning in the 2D Artificial Life System "Avida". In *Artificial Life IV*, pages 377–381. MIT Press.

Banzhaf, W., Baumgaertner, B., Beslon, G., Doursat, R., Foster, J. A., McMullin, B., de Melo, V. V., Miconi, T., Spector, L., Stepney, S., and White, R. (2016). Defining and simulating open-ended novelty: Requirements, guidelines, and challenges. *Theory in Biosciences*, 135(3):131–161.

Bedau, M. A., McCaskill, J. S., Packard, N. H., Rasmussen, S., Adami, C., Green, D. G., Ikegami, T., Kaneko, K., and Ray, T. S. (2000). Open problems in artificial life. *Artificial Life*, 6(4):363–376.

Berlekamp, E. R., Conway, J. H., and Guy, R. K. (1982). *Winning Ways for Your Mathematical Plays, Volume 2: games in particular*. Academic Press.

Clark, E. B., Hickinbotham, S. J., and Stepney, S. (2017). Semantic closure demonstrated by the evolution of a universal constructor architecture in an artificial chemistry. *Journal of the Royal Society Interface*, 14:20161033.

Colizzi, E. S. and Hogeweg, P. (2016). Parasites sustain and enhance RNA-Like replicators through spatial Self-Organisation. *PLoS Comput. Biol.*, 12(4):e1004902.

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

Gardner, M. (1970). Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223(4):120–123.

Greenbaum, B. and Pargellis, A. (2016). Digital Replicators Emerge from a Self-Organizing Prebiotic World. In *ALife 2016*, pages 60–67. MIT Press.

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., Stepney, S., and Hogeweg, P. (2020). Nothing in evolution makes sense except in the light of parasites. (in prep.).

Johnson, T. J. and Wilke, C. O. (2004). Evolution of resource competition between mutually dependent digital organisms. *Artificial Life*, 10(2):145–156.

Maynard Smith, J. and Szathmáry, E. (1995). *The Major Transitions in Evolution*. Oxford University Press.

Pargellis, A. N. (2001). Digital life behavior in the Amoeba world. *Artificial Life*, 7(1):63–75.

Ray, T. S. (1992). An approach to the synthesis of life. In *Artificial Life II*, pages 371–408. Addison-Wesley.

Reynolds, C. W. (1987). Flocks, Herds and Schools: A Distributed Behavioral Model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 25–34. ACM.

Stepney, S. and Hoverd, T. (2011). Reflecting on open-ended evolution. In *ECAL 2011, Paris, France, August 2011*, pages 781–788. MIT Press.

Stepney, S., Polack, F. A. C., Alden, K., Andrews, P. S., Bown, J. L., Droop, A., Greaves, R., Read, M., Sampson, A. T., Timmis, J., and Winfield, A. F. T. (2018). *Engineering Simulations as Scientific Instruments: a pattern language*. Springer.

Takeuchi, N. and Hogeweg, P. (2008). Evolution of complexity in RNA-like replicator systems. *Biol. Direct*, 3:11.

Taylor, T. (2019). Evolutionary Innovations and Where to Find Them: Routes to Open-Ended Evolution in Natural and Artificial Systems. *Artificial Life*, 25(2):207–224.

Taylor, T., Bedau, M., Channon, A., Ackley, D., Banzhaf, W., Beslon, G., Dolson, E., Froese, T., Hickinbotham, S., Ikegami, T., McMullin, B., Packard, N., Rasmussen, S., Virgo, N., Agmon, E., Clark, E., McGregor, S., Ofria, C., Ropella, G., Spector, L., Stanley, K. O., Stanton, A., Timperley, C., Vostinar, A., and Wiser, M. (2016). Open-Ended Evolution: Perspectives from the OEE Workshop in York. *Artificial Life*, 22(3):408–423.

Williams, L. R. (2016). Programs as Polypeptides. *Artificial Life*, 22(4):451–482.