A Self-Replicating System of Ribosome and Replisome Factories

Lance R. Williams[1]

[1]Department of Computer Science, University of New Mexico, Albuquerque, NM 87131
williams@cs.unm.edu

Abstract

An artificial chemistry with composition devices borrowed from object-oriented and functional programming languages was introduced in prior work. *Actors* in *object-oriented combinator chemistry* are embedded in space and subject to diffusion; since they are neither created nor destroyed, mass is conserved. This paper further develops these ideas and applies them in significant ways. First, it introduces the concept of a self-replicating system's *normalized complexity*. Normalized complexity permits comparisons between artificial organisms defined in different virtual worlds by explicitly accounting for the relative complexities of both organism and world. Second, object-oriented combinator chemistry is used to define a parallel, asynchronous, spatially distributed self-replicating system modeled in part on the living cell. This system is strongly constructive since interactions among its parts results in the construction of more of these same parts; constructed parts are assembled from elements of a few primitive types. The system's high normalized complexity is contrasted with that of a simple composome, which is also defined.

Introduction

Much as Turing (1936) had done when motivating his abstract computing machine by comparing it to a human 'computer' executing programs with paper and pencil, von Neumann (1966) began his study of *self-replication in the abstract*, by thinking about a *concrete* physical machine. As imagined, von Neumann's *kinematic automaton* assembled copies of itself from a supply of components undergoing random motion on the surface of a lake. The components consisted of girders, sensors, effectors, logic gates and delays, together with tools for welding and cutting. It is unlikely that von Neumann ever intended to actually *build* a physical self-replicating machine. More likely, he regarded the kinematic automaton as a thought experiment, and abandoned it when he understood how the problems of self-reference, control and construction that truly interested him could be rigorously formulated in the abstract domain of *cellular automata (CA)*.

By abandoning his kinematic automaton, von Neumann became the first 'player' of a sometimes abstruse 'game' that many others have played since (Sipper, 1998). This 'game' has two parts and two pitfalls. Roughly speaking, the parts are: *define a model of computation*; and *define a self-replicating object (or system) in the model.* The two pitfalls, which must be avoided if the 'game' is to be non-trivial are: *making the computational model too abstract*; and *making the primitives too complex.* For example, it is trivial for a self-replicating object defined as a 1 to 'replicate' in an array of 0s if physics is defined to be a Boolean 'or' operation in neighborhoods. It is equally trivial for a self-replicator comprised of a robotic arm, a camera, and a computer to make copies of itself given a supply of robotic arms, cameras and computers. Von Neumann himself was very conscious of the parts and pitfalls and discusses the tradeoffs the 'game' presents at length.[1] His ingenious solution was (characteristically) a saddle point, combining a fiendishly simple model of computation and an enormously complex self-replicating object.

Nearly sixty years after von Neumann's death, no one has yet constructed a kinematic automaton of the kind he imagined. However, because the 'game' seems to many of us to still afford the best prospect by which to address the twin problems of the origin of life on Earth and its evolution into forms of increasing complexity, there is no shortage of new 'players.' Fortunately, current 'players' are the beneficiaries of a wealth of biological science unknown to von Neumann (Watson and Crick, 1953), of significant advances in the science of computing that von Neumann and Turing played seminal roles in founding, and of a growing body of work in the interdisciplinary fields of artificial life and complex systems.

Unsurprisingly (given the preceding), this paper contains descriptions of both a new model of computation and of a self-replicating system defined using that model. The design of *our* model is strongly influenced by the belief that something important was lost when von Neumann adopted cellular automata as *his* model. More specifically, we believe that *conservation of mass*, a law which all machines that assemble copies of themselves from parts must obey, was (in effect) the "baby thrown out with the bath water."

[1]See von Neumann (1966) p. 76-77 and Arbib (1966) p. 179.

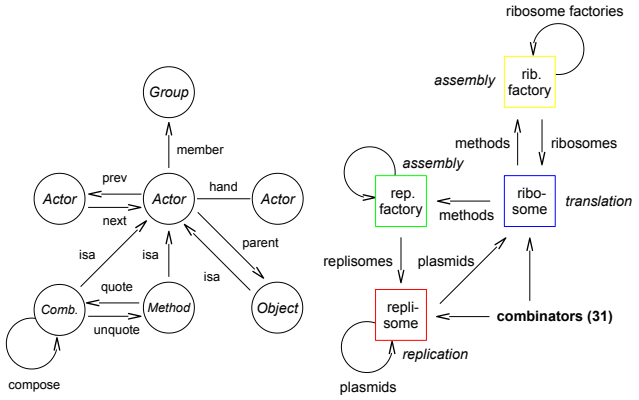

Figure 1: *Actor* datatype in *object-oriented combinator chemistry (OOCC)* (left). Self-replicating system of ribosome and replisome factories (right).

Our starting point is *artificial chemistry* (Dittrich et al., 2001), the study of the population dynamics of systems of constructible objects, which Fontana and Buss (1996) called *constructive dynamical systems*. Like them, we looked to the field of computer science for inspiration, hoping to repurpose elements of modern *object-oriented* and *functional* programming languages as primitives and composition devices in an artificial chemistry. From object-oriented programming we borrowed the ideas of object composition and the association of programs with the data they operate on; from functional programming, we borrowed the idea of construction of programs by composition of 'program fragments' or *combinators.*

The first pitfall (excess abstraction) is avoided using a twofold strategy. First, we make our artificial chemistry concrete by embedding its constructed objects in space and relying solely on diffusion for dynamics. Significantly, to make this tractable, aggregates are treated as *masses* (unlike Arbib (1966) who treated aggregates as *areas*), and mass is conserved, which makes it a more plausible host for a kinematic automaton of the sort imagined by von Neumann. Second, as a guarantee of a different kind of realism, we insist that our artificial chemistry must be a *bespoke physics interface* as defined by Ackley (2013). More specifically, it must function as an abstract interface to a physically realizable *indefinitely scalable* computational substrate. This is consistent with the notion that kinematic automata defined using such interfaces, and which replicate by assembly of conserved parts, are tantamount to physical machines.

We avoid the second pitfall (complex primitives) by using (admittedly) complex primitives to build a self-replicating system comprised of parts that are *still more complex*. However, these more complex parts are constructed by the system itself! It follows that the system is *strongly constructive* in the sense that interactions among its parts result in the construction of more of these same parts (Dittrich et al., 2001). Our inspiration, the ribosome, allowed us to imagine programs as enzymes and to define a pair of representations for programs, one spatially distributed and inert, the other compact and metabolically active. The self-replicating system that resulted is a parallel, asynchronous, distributed computation modeled in part on the living cell. See Figure 1.

The model of computation described in this paper has features in common with prior work on automata and artificial chemistry. The idea of movable aggregates of complex automata has precedent in Arbib (1966). The idea that kinematic automata can be built in embedded artificial chemistries has precedent in Laing (1977), Smith et al. (2003) and Hutton (2004). The use of nested multisets for object composition in an artificial chemistry has precedent in Paun (1998). The use of sequences of combinators as molecules and conservation of mass has precedent in di Fenizio (2000). The idea of molecules as programs has precedent in Laing (1977), Fontana and Buss (1996), di Fenizio (2000) and Hickinbotham et al. (2011). Finally, Taylor (2001) has argued that embeddedness and competition for matter, energy and space are necessary in artificial life systems capable of open-ended evolution.

Normalized Complexity

Pattee (1995) has described the simulation of an organism in a virtual world as an initial value problem where organisms are contingent states subject to the non-contingent laws of physics. In the 'game' of inventing both, there is a tradeoff between the *non-contingent complexity* of models of physical law, and the purely *contingent complexity* of artificial organisms defined inside those models. If physical law is too powerful, self-replication becomes trivial; life is too easy. Conversely, if physical law is not powerful enough, self-replication becomes impossible; life is too hard. It's possible that the most interesting games, those resulting in a bootstrapping process that culminates in organisms capable of open-ended evolution, are grounded in physical law "just powerful enough." Von Neumann's universal replicator R_V and its cellular automata virtual world CA_V suggest that interesting solutions to the 'game' are saddle points, maximizing the ratio of contingent $K(R_V \mid CA_V)$ and non-contingent complexity $K(CA_V)$:

$$\frac{K(R_V \mid CA_V)}{K(CA_V)}$$

where K is Kolmogorov complexity (Kolmogorov, 1965).[2] To explore this idea, let's consider a hierarchy of computational models; each model is built using an interface exposed

[2]Kolmogorov complexity is correct only if the replicator contains no *untranslated information.* Indeed, a pure template replicator, *e.g.,* Smith et al. (2003), might have very high Kolmogorov complexity yet its normalized complexity is zero since it is composed entirely of information that (apart from being copied) is never used.

by a more fundamental model. For example, the problem of simulating a CA with a more fundamental CA is described by Smith (1971). Among many other things, he showed that any CA with a Moore neighborhood (8 neighbors) can be simulated by a CA with a von Neumann neighborhood (4 neighbors) with an increase in space and a slowdown in time by constant factors that depend only on the numbers of states in the CA being simulated:

$$CA_8(\mathbb{Z}^2) \leq_1 CA_4(\mathbb{Z}^2)$$

where CA_8 and CA_4 are CAs with Moore and von Neumann neighborhoods, $\mathbb{Z}^2$ is the integer lattice and $(\leq_1)$ is Smith's $O(1)$ reduction.

Asynchronous cellular automata (ACA) are much like cellular automata except that local state is updated asynchronously (Nehaniv, 2004). It is possible to demonstrate by construction that any CA can be simulated by an ACA with an increase in space (Nakamura, 1974; Nehaniv, 2004) and a slowdown in time (Berman and Simon, 1988) by constant factors that depend only on the number of states and neighborhood size of the CA. Consequently,

$$CA(\mathbb{Z}^2) \leq_1 ACA(\mathbb{Z}^2)$$

where $(\leq_1)$ is Nakamura's $O(1)$ reduction. Our approach is premised on the idea that models in the *object-oriented combinator chemistry (OOCC)* defined in this paper can be compiled to ACAs of one higher dimension. Objects in the artificial chemistry are instances of a recursive datatype grounded in a small number of primitive types and closed under two forms of composition. The extra dimension is used to represent the internal structure of composed objects and the size of this representation is defined as an object's *mass*:

$$OOCC(\mathbb{Z}^2) \leq_1 ACA(\mathbb{Z}^2 \times \mathbb{N})$$

where $(\leq_1)$ is the hypothesized compilation process. Unlike Arbib (1966) who assumed that arbitrarily large automata aggregates could be moved $O(1)$ distance in $O(1)$ time, we assume only that objects of mass m can be moved $O(1)$ distance in $O(m)$ time.

While the significance of our work does not depend on it, the hypothesized compilation process is intriguing because ACAs of dimension three or less can (in principle) be physically realized in hardware. Furthermore, this can be done such that the abstract dimensions of space and time in the ACA (and of all models that have been $O(1)$ reduced to it) are coextensive with physical dimensions of space and time:

$$ACA(\mathbb{Z}^3) \leq_1 U$$

where U is the physical universe. This is the basis for the claim that our artificial chemistry is a bespoke physics interface as defined by Ackley (2013) and that kinematic automata built with it are tantamount to physical machines.

Given a replicator R defined on top of a hierarchy of models reducible to U by $O(1)$ reduction $R \leq_1 M_N \leq_1 \cdots \leq_1 M_1 \leq_1 U$, the ratio of the contingent and non-contingent complexities of replicator R becomes

$$\frac{K(R \mid M_N)}{K(M_N \mid M_{N-1}) + \cdots + K(M_2 \mid M_1) + K(M_1)}.$$

The meaning of this quantity, which will henceforward be termed a replicator's *normalized complexity*, is best illustrated by an example. Codd (1968) was able to significantly simplify von Neumann's replicator and its host CA. Although the contingent complexity of the Codd replicator is significantly less than that of the von Neumann replicator, we speculate that (were they calculated) their normalized complexities would be closer in value.

Langton (1984) defined a much simpler 'loop' replicator L_L on top of the Codd CA substrate. Its contingent complexity, $K(L_L \mid CA_C)$, is much less than that of the Codd replicator, $K(R_C \mid CA_C)$. Nehaniv (2004) showed how the Codd CA substrate could be $O(1)$ reduced to an ACA and demonstrated the Langton 'loop' running on top of the Codd CA running on top of this ACA. These results allow us to compare the normalized complexities of the Codd replicator and the Langton 'loop' defined with respect to the same hierarchy of computational models:

$$\frac{K(L_L \mid CA_C)}{K(CA_C \mid ACA_N) + K(ACA_N)} < \frac{K(R_C \mid CA_C)}{K(CA_C \mid ACA_N) + K(ACA_N)}.$$

Hutton's work on "artificial cells" provides a second example (Hutton, 2004). In Hutton's virtual world, physical law takes the form of an artificial chemistry defined by a set of 34 graph rewrite rules. Hutton's artificial organism is a cell-like configuration of atoms $C_H + P_1$ containing a small (non-functional) information payload P_1. Significantly, Hutton demonstrated that both the 'cell' and its payload are replicated by the 'reaction' rules of the artificial chemistry. Because the payload P_1 is untranslated, the contingent complexity of Hutton's cell is $K(C_H \mid AC_{34})$.

Hutton subsequently extended AC_{34} by adding six rules for translating the payload P_1 into an 'enzyme' E_1 capable of 'catalyzing' an arbitrary reaction and used this enzyme to replace one of the graph rewrite rules, R_1. In doing so, the (non-functional) information payload becomes a (functional) partial genome and some part of the system's complexity moves from the non-contingent to the contingent category. However, this exchange is insufficient to offset the increase in non-contingent complexity that results from the addition of the six rules. Consequently,

$$\frac{K(E_1 \mid P_1,\, AC_{40} - R_1) + K(C_H + P_1 \mid AC_{40} - R_1)}{K(AC_{40} - R_1)} < \frac{K(C_H \mid AC_{34})}{K(AC_{34})}.$$

where $K(E_1 \mid P_1,\, AC_{40} - R_1)$ is zero since P_1 encodes E_1 using a process defined by the artificial chemistry $AC_{40} - R_1$.

Object-Oriented Combinator Chemistry

There are three types of actors: *objects*, *methods* and *combinators*. Objects and methods are like objects and methods in object-oriented programming. More specifically, objects are containers for actors, methods are programs that govern actors' behaviors, and combinators are the building blocks used to construct methods (see Figure 1). Like amino acids, which can be composed to form polypeptides, *primitive* combinators can be composed to form *composite* combinators. A method is just a composite combinator that has been repackaged or *unquoted*. Prior to unquoting, combinators do not manifest behaviors, so unquoting might correspond (in this analogy) to the folding of a polypeptide chain into a protein.

Objects are *multisets* of actors. They are of four immutable types constructed using $\{\}_1$, $\{\}_2$, $\{\}_3$ and $\{\}_4$. For example, $\{x, y, z\}_2$ is an object of type two that contains three actors, x, y and z. Combinators are composed with $(>\!\!=\!\!>)$ and quoted and unquoted using $()^-$ and $()^+$. Primitive combinators and empty objects have unit *mass.* The mass of a composite combinator is the sum of the masses of the combinators of which it is composed. The mass of an object is the sum of its own mass and the masses of the actors it contains. Since actors can neither be created nor destroyed, mass is conserved.

Actors are *reified* by assigning them positions in a 2D virtual world. Computations progress when actors interact with other actors in their Moore neighborhoods by running methods. Methods are sequences of combinators compiled from programs defined in a visual programming language. The programs in this language, *dataflow graphs*, serve as abstract specifications of actors' behaviors. Neither the visual programming language nor the combinator language are described in this paper since both were described at length in Williams (2015).

All actors are subject to *diffusion*. An actor's diffusion constant decreases inversely with its mass. This reflects the real cost of data transport in the (notional) $ACA(\mathbb{Z}^2 \times \mathbb{N})$ substrate. Multiple actors can reside at a single site, but diffusion never moves an actor to an adjacent occupied site if there is an adjacent empty site.

The object that contains an actor (with no intervening objects) is termed the actor's *parent.* An actor with no parent is a *root*. Root actors (or actors that have the same parent) can associate with one another by means of *groups* and *bonds.* Association is useful because it allows working sets of actors to be constructed and the elements of these working sets to be addressed in different ways.

The first way in which actors can associate is as members of a *group*. All actors belong to exactly one group and this group can contain a single actor. For this reason, the group relation is an *equivalence relation* on the set of actors. A group of root actors is said to be *embedded*. All of the actors in an embedded group diffuse as a unit and all methods run by actors in an embedded group (or contained inside such actors) share a finite time resource in a zero sum fashion. Complex computations formulated in terms of large numbers of actors running methods inside a single object or group will therefore be correspondingly slow. Furthermore, because of its large net mass, the object or group that contains them will also be correspondingly immobile.

The second way in which actors can associate is by *bonding*. Bonds are short relative addresses that are automatically updated as the actors they link undergo diffusion. Because bonds are short (L_1 distance less than or equal to two), they restrict the diffusion of the actors that possess them. Undirected bonds are defined by the *hand* relation H, which is a *symmetric relation* on the set of actors, *i.e.,* $H(x,y) = H(y,x)$. Directed bonds are defined by the *previous* and *next* relations, P and N, which are *inverse relations* on the set of actors, *i.e.,* $P(x,y) = N(y,x)$. An actor can possess at most one bond of each type.

Apart from composition, containment, groups and bonds there is no other mutable persistent state associated with actors. In particular, there are no integer registers. Primitive combinators exist for addressing individual actors or sets of actors using most of these relations. Other primitive combinators modify actors' persistent states.

Composomes

Composomes are quasi-stationary molecular assemblies that preserve *compositional information* (Segré et al., 2000). As self-replicating entities, they possess very low normalized complexity because they do not construct the parts of which they are comprised, and individually, these parts are more complex than the composome itself. Nevertheless, a composome serves as a good first example, and we can construct one by defining a set of behaviors using dataflow graphs and reifying them as an embedded group of methods:

$$X = \{\text{cmpA}, \text{cmpB}, \text{cmpC}\}$$

where cmpA, cmpB and cmpC are the three methods and $\{\}$ denotes an embedded group. The composome's first two methods run in the mother group (the group being copied) while its third runs in the daughter group (the copy). Because methods contained in different embedded groups run in parallel, they do not compete for cycles; this decreases the time required for self-replication.

- If cmpA is in a group with others but no members of its group have bonds then it finds an unbonded actor in its neighborhood similar to itself with no others in its group and creates a *next* bond with it.

- CmpB first verifies it is in the mother group. If it also has an unbonded neighbor similar to a member of its group and if the neighbor is not already in a group then it adds the neighbor to the daughter's group.

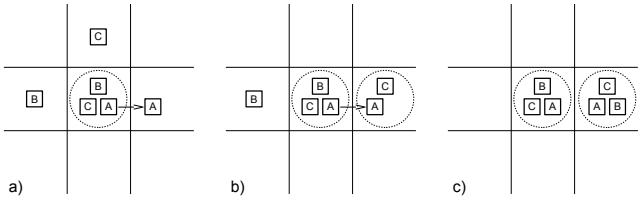

Figure 2: Self-replication by *composome*. a) CmpA forms a *next* bond with another cmpA instance in its neighborhood. b) CmpB finds a cmpC instance in its neighborhood and adds it to the daughter group. c) CmpC (in the daughter group) deletes the bond joining the mother and daughter groups after cmpB is added.

- CmpC first checks to see if it is in the daughter group. It does this by verifying that there is a group member with a *prev* bond (which can only be of type cmpA). It then verifies that there is also a group member not similar to either the cmpA instance or itself. By process of elimination, this group member must be of type cmpB. Since the daughter group contains the complete set of methods, it can delete the *prev* bond which joins the cmpA instances of the mother and daughter groups.

When placed into the virtual world with a supply of the methods that comprise it, the following reaction occurs

$$X + \text{cmpA} + \text{cmpB} + \text{cmpC} \rightarrow 2X.$$

Because cmpB does not check to see whether the composome already possesses a method before adding it to the daughter group, the fraction of reactants converted to complete composomes (self-replication efficiency) will be significantly less than 100%.

Ribosomes

Biological enzymes can be reified as chains of nucleotides or amino acids. The first can be read and copied but are spatially distributed and purely representational; the second are representationally opaque but compact and metabolically active. Dataflow graphs can be compiled into sequences of primitive combinators and reified in analogous ways: *genes* can be read and copied but do not manifest behaviors; *enzymes* manifest behaviors but cannot be read or copied. A *gene* is a spatially extended chain of actors of type combinator linked with *directed* bonds:

$$G_i = >_{j=1}^{|G_i|} c_i(j)$$

where $c_i(j)$ is combinator j of gene i and $(>)$ are directed bonds. As in the genomes of living cells, sets of genes that are expressed together can be grouped together. A *plasmid* is a sequence of genes joined with *undirected* bonds:

$$P = |_{i=1}^{|P|} >_{j=1}^{|G_i|} c_i(j)$$

where $(|)$ are undirected bonds. An additional undirected bond $c_{|P|}(\,|G_{|P|}|\,) \mid c_1(1)$ closes the chain. While plasmids are spatially distributed chains of multiple actors, *enzymes* are single actors of type method:

$$E_i = (\ >\!\!\Rightarrow_{j=1}^{|G_i|}\ c_i(j)\ )^+$$

where $(>\!\!\Rightarrow)$ is Kleisli composition and $(\,)^+$ is the constructor for actors of type method. In addition to plasmids, comprised of genes, a minimum self-replicating system might contain objects of three types. *Ribosomes* translate genes into enzymes and *replisomes* copy plasmids. *Factories* are copiers of compositional information, namely, the sets of enzymes and objects that comprise ribosomes, replisomes and factories themselves. A self-replicating system like this would possess *semantic closure* (Pattee, 1995) because it would construct the parts that comprise it (enzymes) from descriptions contained within itself (genes). Unlike the cell (where enzymes are sequences of amino acids and genes are sequences of nucleotides) enzymes and genes are built from the same elementary building blocks, *i.e.,* combinators.

Biological ribosomes translate descriptions of proteins encoded as sequences of nucleotides into polypeptides, sequences of amino acids, the building blocks of proteins. A *computational ribosome* translates a plasmid into one or more enzymes by traversing genes while composing combinators from the neighborhood matching those comprising the gene. In functional pseudocode, the ribosome evaluates the following expression:

$$\text{map}_{|}\ ((\,)^+ \cdot\ (\text{fold}_{>}\ (>\!\!\Rightarrow\!))))\ P$$

where $map_{|}$ maps functions over the genes G_i that comprise plasmid P and $fold_{>}$ is right fold over the combinators $c_i(j)$ that comprise a gene. A computational ribosome can be constructed by defining four enzymes that perform these functions and placing them inside an actor of type object:

$$R = \{\text{ribA, ribI, ribE, ribT}\}_0.$$

RibA first attaches R to the plasmid by adding it to the group of the initial combinator of some gene, $c_i(1)$. Afterwards, the (now unnecessary) ribA is expelled (and R becomes R'); see Figure 3 (a).

After ribosome attachment, ribI finds an actor in the neighborhood with type matching $c_i(1)$ and places it inside R'; see Figure 3 (b). When R' is at position j on the plasmid, ribE finds a neighbor with type matching $c_i(j+1)$ and composes it with the combinator contained in R', *i.e.,* with $c_i(1) >\!\!\Rightarrow \cdots >\!\!\Rightarrow c_i(j)$. It then advances the position of R' to $j+1$ by following the *next* bond; see Figure 3 (c). This process continues until R' reaches the last combinator in the gene, $c_i(\,|G_i|\,)$, which possesses a *hand* bond, at which point ribT promotes the combinator to a method, expels the method, and moves the ribosome across the bond.

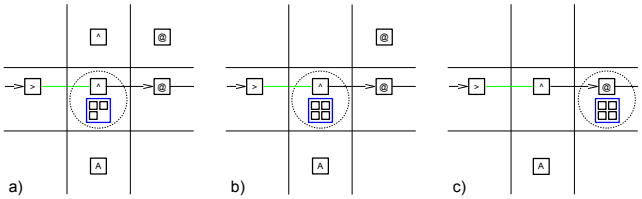

Figure 3: a) Ribosome attaches itself to plasmid at gene origin (marked by *hand* bond) and ejects ribA. b) Combinator from neighborhood matching initial combinator is placed inside ribosome. c) Combinator from neighborhood matching next combinator of plasmid is composed with combinator inside ribosome and ribosome advances.

If plasmid P and ribosome R are placed in the virtual world with a supply of primitive combinators $\sum_P \sum_C h(p,c)\, c$ then the ribosome manufactures the enzymes $\sum_P E_p$ described by the plasmid

$$P + R + \sum_P \sum_C h(p,c)\, c \rightarrow P + R' + \text{ribA} + \sum_P E_p$$

where C is the set of 42 primitive combinators and $h(p,c)$ is the number of combinators of type c in G_p and E_p, *i.e.,* the gene and enzyme reifications of behavior p.

Replisomes

We have already defined a computational ribosome, *i.e.,* an object that translates inert descriptions of behaviors encoded by a plasmid (genes) into behaviors reified as methods able to do actual work (enzymes). We now turn our attention to the problem of defining a *computational replisome*, an object that will replicate plasmids. In functional pseudocode, the replisome evaluates the following expression:

$$(\text{fold}_{|}\ (\,|\,) \cdot \text{map}_{|}\ (\text{fold}_{>}\ (>)))\ P$$

where $(\,|\,)$ and $(>)$ are functions that create undirected and directed bonds, $fold_{|}$ is right fold over the genes G_i that comprise plasmid P and $fold_{>}$ is right fold over the combinators $c_i(j)$ that comprise a gene.

Biological replisomes copy plasmids in pairs. Replication begins when two replisomes are assembled at the plasmid's *replication origin*. Each replisome manages one *replication fork*. The replication forks move away from the replication origin in opposite directions and replication is finished when the pair of replisomes reunite at a position on the plasmid opposite the origin. A computational replisome can be designed that works in a similar way. As in a cell, there are two replication forks. However, unlike a cell, only one moves; the other is stationary. The replisome manages the active replication fork. It is initially an object containing five enzymes and an empty object of the same type as itself:

$$Q = \{\text{repA, repE, repF, repY, repZ, } \{\,\}_2\,\}_2.$$

RepA first causes the replisome to attach to the plasmid. It does this by adding the replisome to the group of one of the plasmid's combinators. Afterwards, the stationary replication fork is marked by attaching the empty object $\{\,\}_2$ contained within the replisome to the combinator that precedes the replisome's own attachment site. RepA creates a directed bond from the marker to the replisome and then ejects itself since it is no longer needed; see Figure 4 (b).

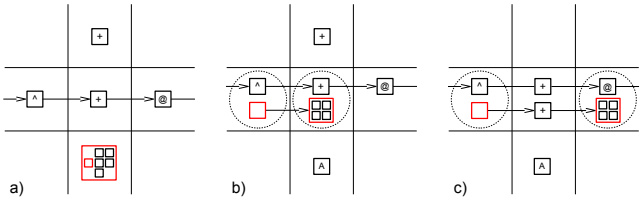

Figure 4: a) A short segment of a plasmid and a replisome containing five enzymes and an empty object *marker* of the same type as itself. b) Replisome attaches itself to the plasmid by joining the group of one of its combinators; attaches the marker to the combinator preceding its own attachment site; forms a directed bond with the marker; and ejects repA. c) Replisome advances along the plasmid after splicing a combinator of the correct type from the neighborhood into the directed bond that trails it. This is the first combinator of the daughter plasmid.

RepE and repF govern the motion of the replication fork. RepE finds a combinator in the neighborhood matching the combinator attached to replisome Q'. It moves Q' in the increasing direction (by joining the group of the combinator that follows the replisome's own attachment site) and splices the neighbor into the growing chain (the incomplete *daughter* plasmid) that trails it; see Figure 4 (c). RepF is very similar except that it moves the replication fork through the *hand* bonds that mark the boundaries between genes.

Replication is complete when Q' encounters a marker, or more precisely, when it finds a marker attached to the combinator that follows its own attachment site. In the most common case, the replisome and marker are situated within a single gene. RepY recognizes this situation and creates the final *next* bond, completing the daughter plasmid. Very infrequently, the replisome and marker straddle a boundary between two genes. RepZ recognizes this situation and creates the final *hand* bond. In both cases, the replisome and marker are detached from the plasmid.

If plasmid P and replisome Q are placed in the virtual world with a supply of primitive combinators $\sum_P \sum_C h(p,c)\, c$ then the replisome copies the plasmid

$$P + Q + \sum_P \sum_C h(p,c)\, c \rightarrow 2\,P + Q' + \text{repA} + \{\,\}_2.$$

Self-Replicating Ribosome and Replisome Factories

Abstractly, factories are copiers of *compositional information*, which is heritable information distinct from the *genetic information* copied by replisomes and which ribosomes translate into enzymes. Concretely, factories are ob-

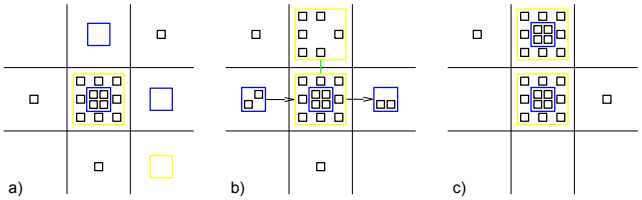

Figure 5: a) Self-replicating ribosome factory contains eight enzymes and a model ribosome. b) Directed bonds connect the factory to partially assembled ribosomes while an undirected bond connects it to a partially assembled daughter factory. c) One of the product ribosomes becomes the model for the daughter factory while the second is available to synthesize enzymes for the encompassing system.

jects containing a specific set of enzymes and a *model*, which can be either a ribosome or a replisome; see Figure 5 (a). A factory's enzymes can be grouped into four categories:

- FacP, facN and facH form *prev*, *next* and *hand* bonds with empty objects from the neighborhood. The order in which these three events occur is more or less random. The objects $\{\}_k$ bonded to the mother factory by *prev* and *next* bonds will become new instances of the model. The object $\{\}_{k+1}$ bonded to the mother factory by the *hand* bond will become the daughter factory; see Figure 5 (b).
- FacU moves enzymes with types matching contents of the model into the incomplete model instances. FacV moves enzymes with types matching contents of the mother factory into the incomplete daughter factory.
- FacX uses the generalized set difference operator to verify that a new model instance, *i.e.*, *product*, has all of the enzymes that the old model contains. If so, it marks the product as complete using a self-directed *hand* bond. FacY does the same thing for the daughter factory but uses a self-directed *prev* bond to indicate completeness.
- FacZ checks to see that both products have self-directed *hand* bonds and also that the daughter factory has a self-directed *prev* bond. If so, it 1) deletes the *prev* and *next* bonds connecting the mother factory and the products; 2) moves one of the completed products into the daughter factory (to serve as its model); and 3) deletes the *hand* bond connecting the mother and daughter factories; see Figure 5 (c).

Given the above enzymes, it is now possible to define a self-replicating ribosome factory:

$$F_R = \{\text{facP, facN, facH, facU, facV, facX, facY, facZ}, R\}_1.$$

When placed in the virtual world with a supply of empty objects and enzymes comprising ribosomes $\sum_R E_r$ and factories $\sum_F E_f$ the ribosome factory constructs a new ribosome factory and a new ribosome:

$$F_R + 2\{\}_0 + \{\}_1 + 2\sum_R E_r + \sum_F E_f \rightarrow 2F_R + R.$$

A self-replicating replisome factory F_Q can be defined similarly. When placed in the virtual world with a supply of empty objects and enzymes comprising replisomes $\sum_Q E_q$ and factories $\sum_F E_f$ the replisome factory constructs a new replisome factory and a new replisome:

$$F_Q + 4\{\}_2 + \{\}_3 + 2\sum_Q E_q + \sum_F E_f \rightarrow 2F_Q + Q.$$

Note that the left hand side of the reaction contains four empty objects $\{\}_2$ instead of two; the extras are the markers contained by the new replisome and replisome model.

We now have all of the components needed to build a self-replicating system of ribosome and replisome factories. Unlike the composome, which copied itself solely by reflection, the self-replicating system is a *quine* that translates and replicates a self-description reified as a data structure within the virtual world itself:

$$P_{17} = ribA \mid ribI \mid ribE \mid ribT \mid repA \mid repE \mid repF \mid repY \mid repZ \mid facP \mid facN \mid facH \mid facU \mid facV \mid facX \mid facY \mid facZ.$$

This *genome* consists of a single plasmid containing 586 combinators comprising 17 genes. The minimum *phenome* required for bootstrapping the self-replicating system consists of a replisome factory F_Q, a ribosome factory F_R and a ribosome R. When genome P_{17} and phenome $F_Q + F_R + R$ are placed in the virtual world with a supply of empty objects $\{\}_k$ and combinators $\sum_{P_{17}} \sum_C h(p,c)\,c$, the system increases the redundancy of all of its component parts:

$$P_{17} + F_Q + F_R + R + 2\{\}_0 + \{\}_1 + 4\{\}_2 + \{\}_3 + 3\sum_{P_{17}} \sum_C h(p,c)\,c \rightarrow 2P_{17} + 2F_Q + 2F_R + Q' + \text{repA} + \{\}_2 + R + R' + \text{ribA}.$$

Note that there are three instances of $\sum_{P_{17}} \sum_C h(p,c)\,c$ on the left side of the equation. The ribosome R consumes the first two making two full circuits of the plasmid synthesizing the system's enzymes

$$P_{17} + R + 2\sum_{P_{17}} \sum_C h(p,c)\,c \rightarrow P_{17} + R' + \text{ribA} + 2\sum_{P_{17}} E_p$$

while the replisome Q (assembled by F_Q) uses the last copying the plasmid.

Comparison of Normalized Complexities

It is useful to compare the normalized complexities of the self-replicating system of ribosome and replisome factories and the composome defined earlier. Recall that the composome X is composed of 3 enzymes of 3 types: cmpA, cmpB and cmpC; these enzymes are in turn composed of 66 combinators of 17 types. Because the enzymes are defined outside the system, their complexity is non-contingent, and the composome's normalized complexity is quite low:

$$\frac{K(\{\text{cmpA, cmpB, cmpC}\} \mid \text{cmpA} + \text{cmpB} + \text{cmpC})}{K(\text{cmpA} + \text{cmpB} + \text{cmpC} \mid OOCC_{17}) + K(OOCC_{17} \mid ACA) + K(ACA)}$$

where $K(\text{cmpA} + \text{cmpB} + \text{cmpC} \mid OOCC_{17})$ is the portion of the composome's non-contingent complexity contained in its three enzymes.

In contrast, the self-replicating system of ribosome and replisome factories is composed of 17 different behaviors reified as both genes and enzymes; these genes and enzymes are in turn composed of $3 \times 586 = 1758$ combinators of 31 types. Furthermore, because the enzymes are defined within the system itself (by the genes), their complexity is (unlike that of the composome's enzymes) contingent. Consequently, the self-replicating system of ribosome and replisome factories possesses significantly higher normalized complexity than the composome:

$$\frac{K(F_Q + F_R + R \mid \sum_{P_{17}} E_p) + K(\sum_{P_{17}} E_p \mid P_{17}, R, OOCC_{31}) + K(P_{17} \mid OOCC_{31})}{K(OOCC_{31} \mid ACA) + K(ACA)}$$

where $K(F_Q + F_R + R \mid \sum_{P_{17}} E_p)$ and $K(P_{17} \mid OOCC_{31})$ are the compositional and genetically encoded portions of the self-replicating system's contingent complexity and $K(\sum_{P_{17}} E_p \mid P_{17}, R, OOCC_{31})$ is zero because the plasmid P_{17} encodes the enzymes $\sum_{P_{17}} E_p$ using a process defined by the ribosome R and the object-oriented combinator chemistry.

Conclusion

This paper introduced the concept of a self-replicating system's *normalized complexity*. Normalized complexity permits comparisons between artificial organisms defined in different virtual worlds by explicitly accounting for the relative complexities of both organism and world. An *object-oriented combinator chemistry* (introduced in prior work) was used to define a parallel, asynchronous, spatially distributed self-replicating system modeled in part on the living cell. The high normalized complexity of this self-replicating system of ribosome and replisome factories was contrasted with that of a simple composome.

References

Ackley, D. (2013). Bespoke physics for living technology. *Artificial Life*, 34:381–392.

Arbib, M. A. (1966). Simple self-reproducing universal automata. *Information and Control*, 9(2):177–189.

Berman, P. and Simon, J. (1988). Investigations of fault-tolerant networks of computers. In *STOC*, pages 66–77.

Codd, E. F. (1968). *Cellular automata*. Academic Press, London.

di Fenizio, P. S. (2000). A less abstract artificial chemistry. In *Proc. of the 7th Intl. Conf. on the Simulation and Synthesis of Living Systems (ALIFE)*, pages 49–53.

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

Fontana, W. and Buss, L. W. (1996). *The barrier of objects: From dynamical systems to bounded organizations*. International Institute for Applied Systems Analysis.

Hickinbotham, S., Clark, E., Stepney, S., Clarke, T., Nellis, A., Pay, M., and Young, P. (2011). Molecular microprograms. In *European Conference on Artificial Life (ECAL)*, pages 297–304.

Hutton, T. J. (2004). A functional self-reproducing cell in a two-dimensional artificial chemistry. In *Proc. of the 9th Intl. Conf. on the Simulation and Synthesis of Living Systems (ALIFE)*, pages 444–449.

Kolmogorov, A. (1965). Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1(1):1–7.

Laing, R. A. (1977). Automaton models of reproduction by self-inspection. *Journal of Theoretical Biology*, 66(1):437–456.

Langton, C. G. (1984). Self-reproduction in cellular automata. *Physica D: Nonlinear Phenomena*, 10(1):135–144.

Nakamura, K. (1974). Asynchronous cellular automata and their computational ability. *System Comput. Controls*, 15(5):56–66.

Nehaniv, C. L. (2004). Asynchronous automata networks can emulate any synchronous automata network. *IJAC*, 14(5-6):719–739.

Pattee, H. (1995). Evolving self-reference: Matter, symbols, and semantic closure. *Communication and Cognition - Artificial Intelligence*, 12:9–27.

Paun, G. (1998). Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143.

Segré, D., Ben-Eli, D., and Lancet, D. (2000). Compositional genomes: Prebiotic information transfer in mutually catalytic noncovalent assemblies. *Proceedings of the National Academy of Sciences*, 97(8):4112–4117.

Sipper, M. (1998). Fifty years of research on self-replication: An overview. *Artificial Life*, 4(3):237–257.

Smith, A., Turney, P. D., and Ewaschuk, R. (2003). Self-replicating machines in continuous space with virtual physics. *Artificial Life*, 9(1):21–40.

Smith, A. R. (1971). Cellular automata complexity trade-offs. *Information and Control*, 18(5):466–482.

Taylor, T. (2001). Creativity in evolution: Individuals, interactions and environments. In Bentley, P. and Corne, D., editors, *Creative Evolutionary Systems*, pages 79–108. Morgan Kaufman.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *J. of Math*, 58(345-363):5.

von Neumann, J. (1966). Theory of self-replicating automata. *Urbana: University of Illinois Press*.

Watson, J. D. and Crick, F. H. (1953). Molecular structure of nucleic acids. *Nature*, 171(4356):737–738.

Williams, L. R. (2015). Programs as polypeptides. In *European Conference on Artificial Life (ECAL)*, pages 150–157, York, England.