Artificial Cells as Reified Quines

Lance R. Williams[1]

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

Abstract

Cellular automata were initially conceived as a formal model to study self-replicating systems. Although reproduction by biological cells is characterized by exponential population increase, no population of self-replicating machines modeled as a cellular automaton has ever exhibited such rapid growth. We believe this is due to the inability of cellular automata to model bonded complexes of *reified actors* undergoing random independent motion.

To address this limitation, we introduce a model of parallel distributed spatial computation which is highly expressive, indefinitely scalable, and asynchronous. We then use this model to define two examples of self-replicating *kinematic automata*. These machines assemble copies of themselves from components supplied by diffusion and increase in number exponentially until the supply of components is depleted. Because they are both programmable constructors and self-descriptions, we call them *reified quines*.

Introduction

Much as Turing had done twenty years earlier when motivating his computing machine by first describing a notional human computer which computed with paper and pencil (Turing, 1936), von Neumann motivated his self-replicating machine by means of a thought experiment (Burks, 1970). von Neumann's machine assembled copies of itself from a set of components undergoing random independent motion on the surface of a lake. The components consisted of girders, hands, muscles, sensors, switches (*and*, *or* and *not* gates), and delays, together with tools for welding and cutting. von Neumann ultimately concluded that the physics of his machine was too removed from reality to be interesting, while unnecessarily complicating the study of the information processing problems inherent in self-replication. Accordingly, the bulk of his subsequent efforts were concerned with abstract machines not physical machines, and the class of abstract machine he adopted, *cellular automata*, has dominated the field for the past fifty years.

Although self-replication by biological cells is characterized by exponential population increase, no population of self-replicating machines modeled as a cellular automaton has ever displayed such rapid growth. Indeed, populations of the most fecund (Langton, 1984) grow only as a quadratic function of time. We believe this is due to the inability of cellular automata to model bonded complexes of *reified actors* undergoing random independent motion.

Random independent motion, or *diffusion*, plays a crucial role in our work. First, as in von Neumann's kinematic model, components required for self-replication are supplied by diffusion. Second, diffusion changes the length of bonds, and vital operations must wait until bonds are of sufficient length. Third, the products of self-replication are dispersed by diffusion, which is essential for exponential population growth because it prevents overcrowding.

Quines

Self-replicating machines can be divided into two types. The *Darwinian* type contain a self-description (genotype) and replicate by both copying it (yielding a copy of the genotype) and decoding it (yielding a copy of the phenotype). In contrast, the *Lamarckian* type replicate by copying the phenotype directly. Computer *worms* are Lamarckian, while *quines* (programs written in high-level languages which print themselves) are Darwinian. Worms don't need a self-description because of the nearly unique capacity for *reflection* possessed by machine language programs running on digital computers with von Neumann architectures. Programs and data reside in the same memory; programs *are* data. In contrast, most high-level programming languages lack the capacity for reflection. It follows that quines, like biological cells, must replicate by copying and decoding self-descriptions.

Prior Work

The prior work with goals and approach most similar to our own is that of Hutton (2007), who has developed an artificial cell with a membrane in a 2D artificial chemistry. Hutton's cell consists of a membrane formed from a ring of 14 atoms internally bisected by a string of 5 atoms which serves as a partial genome. The membrane is permeable to unbonded atoms but impermeable to bonded atoms. The entire structure is copied atom-by-atom, through the action of 39 reaction rules which define a universal chemistry. Atoms are of

6 different types and can possess up to 62 states each. The reaction rules have a very restricted form; both left and right hand sides consist of a single pair of atoms (either bonded or unbonded), and each in a specified state.

The most impressive aspect of Hutton's work is the partial genome. This is an arbitrary string of atoms which can be used to encode any reaction rule. It is translated into a bonded pair of atoms which functions as an enzyme. Because it is contained inside a membrane impermeable to bonded atoms, it is hoarded by the cell for its exclusive use. Although enzymes can (in principle) be used to replace any of the reaction rules in the artificial chemistry (the single exception presumably being the rule governing the use of enzymes), this has only been demonstrated for a single reaction rule and Hutton (2005) states that a genome 700 atoms long (and a correspondingly larger membrane), would be needed to replace the full set.

Actor Model

Biological cells are membranes made of lipids which contain water, enzymes, and DNA. The DNA encodes the enzymes and the enzymes (in water) form metabolic pathways which collectively: 1) copy the DNA; 2) translate the DNA into enzymes; and 3) make the membrane grow and divide. In our view, biochemistry is parallel distributed computation and enzymes are *actors*. Membranes don't just concentrate and isolate enzymes, they define private *absolute address* spaces. In effect, they permit the construction of idiosyncratic biochemistries, defined by specific sets of enzymes, the descriptions of which are encoded by the cells' own DNA.

The *actor model* is a model of parallel distributed computation (Hewitt et al., 1973). An actor is a process which possesses a unique absolute address. Using these addresses, actors send and receive messages to and from other actors. In response to receiving a message, actors can change state, create new actors, and send new messages. Significantly, and unlike cellular automata, computation in the actor model is event-driven and asynchronous.

With respect to the goal of constructing reified quines, the actor model has a number of shortcomings. First, because of its use of absolute addresses, it is not indefinitely scalable; in an actual implementation, the average time required to deliver a message increases as the number of actors increases. Second, there is no satisfactory method to generate guaranteed unique addresses in a parallel distributed manner. Third, and most significantly, the actor model is not reified–actors exist in an abstract space, not in a space which is isomorphic to physical space.

Reified Actor Model

Although as originally conceived, actor models are not reified, it is possible to create a reified actor model or *movable feast* (Ackley and Cannon, 2011). In a movable feast, all actors have unique positions on a 2D grid. Actors possess a finite number of states and can sense and change the positions and states of actors in their $n \times n$ neighborhoods. Significantly, actors can create *bonds* with other actors in their $n \times n$ neighborhoods. Bonds are *relative addresses* which are short, symmetric, and automatically updated as actors undergo random independent motion (restricted by the lengths of bonds).

The set of actors reachable through a sequence of bonds of length less than or equal to k comprise an actor's *bond graph k-neighborhood*. Actors can sense and change the positions and states of actors in their bond graph k-neighborhoods.

Like conventional actor models, computations in a movable feast are event-driven and asynchronous. Unlike conventional actor models, movable feast computations are based on the application of graph rewrite rules possessed by individual actors to the actors' bond graph k-neighborhoods. Sets of related graph rewrite rules are grouped into *behaviors*, which are indivisible and conferred as units. Actors can *possess* multiple behaviors but can *denote* at most one behavior. Significantly, an actor can confer the behavior it denotes on other actors through bonds. The distinction between possessing and denoting mirrors the phenotype-genotype distinction in biological cells and the program-data dichotomy in quines.

The update scheme in the movable feast consists of picking an actor at random, picking a behavior possessed by that actor at random, and applying the first graph rewrite rule with a pattern matching the actor's bond graph k-neighborhood.

Kinematic Automata

The vertices of a *bond graph* are actors and the edges are bonds; both actors and bonds can be of one or more types. Because they are reified, actors have unique positions on a 2D grid. In homage to von Neumann, we define a *kinematic automaton (KA)* to be a set of reified actors possessing type specific behaviors assembled in a bond graph.

A *description* of a KA consists of a bond graph and a *behavior graph.* The behavior graph represents the relation between the set of types and the set of behaviors, *i.e.,* the *behavior relation*. Actors are finite state machines with transition functions defined by the behaviors they possess (Fig. 1). It follows that a KA is an asynchronous network of communicating finite state machines (Brand and Zafiropulo, 1983); the set of behaviors possessed by its actors define a graph rewriting system (Klavins et al., 2004) which transforms the embedding and topology of the network over time.

A *programmable constructor* for a class of KA's is a KA which takes a description of a KA in the class and builds it. Example classes are *reified-strings* and *reified-sets*. A programmable constructor may (or may not) be in the class it builds. A *self-description* is a KA where the bond graph represents the behavior graph using an encoding scheme; it is

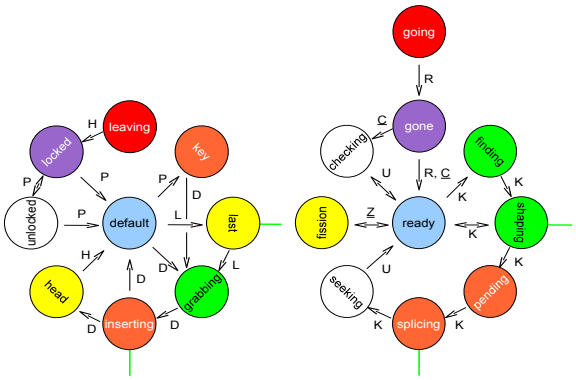

Figure 1: State transition diagrams for actors in reified-string (left) and reified-set (right) quines. Letters denote behaviors mediating state transitions. Green sticks mark states where the actor possesses a *hand* bond.

this use of dual meaning which resolves the seeming paradox of self-description–how can a thing contain a description of itself?

Reified-String Quine

A *reified-string* is a KA consisting of a chain of reified actors linked by bonds. Apart from the head (tail) each actor in the reified-string has a unique predecessor (successor) to which it is bonded by a *prev* bond (*next* bond). A behavior graph can be represented using an adjacency list representation which in turn can be represented as a string. For example, let H, D, P, and L be types denoting behaviors and let # be a punctuation type, then $Q = \#HDP\#DDP\#PDP\#\#HDP\#LDPLL$ is a reified-string self-description where actors of all types possess behaviors D and P while actors of type # also possess behavior H and actors of type L also possess behavior L (the repeated L marks the end of the string). A reified-string self-description which is also a programmable constructor for the class of reified-strings is a *reified-string quine*.

Behaviors

To build a reified-string quine we must define a set of graph rewrite rules which when grouped into behaviors H, D, P, and L yield a Q which is a programmable constructor for reified-strings:

H – initiates decoding phase using *tip*
D – copies string using *grab*, *insert* and *transport*
P – confers type specific behaviors by decoding string using *key*, *lock*, *unlock* and *confer*
L – initiates copying phase, assembles daughter, and effects fission using *cleave*.

The reified-string quine copies itself in two phases. During the *copying* phase, the bond graph is copied actor-by-actor. During the *decoding* phase, the adjacency list representation

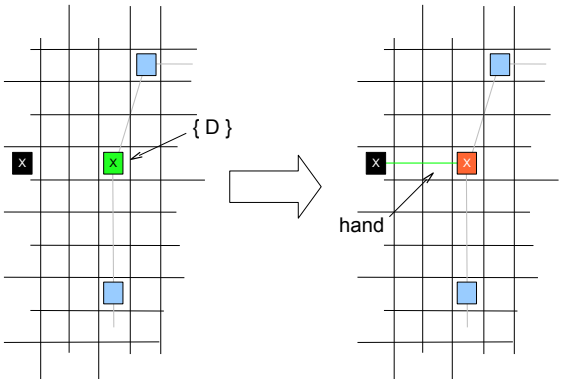

Figure 2: *Grab* graph rewrite rule. An actor in the *grabbing* state possessing behavior D and denoting behavior X forms a *hand* bond with an unbonded actor denoting the same behavior in its $n \times n$ neighborhood. It then enters the *inserting* state.

of the behavior graph is decoded, conferring the behaviors specific to each type on the copies.

Copying

Copying begins at the tail of the reified-string and advances towards the head. *Grab* and *insert* rewrite rules from behavior D cause each actor to

- form a *hand* bond to an unbonded actor of matching type in its $n \times n$ neighborhood (Fig. 2)
- set that actor's state to *leaving*
- insert it into the reified-string nearer the head (Fig. 3).

In effect, the hand advances towards the head as each actor in the mother cycles through the *default*, *grabbing* and *inserting* states. Meanwhile, the *transport* graph rewrite rule from behavior D swaps actors in the *default* state with actors nearer the tail in the *leaving* state, an action which quickly moves them to the head. At the completion of the copying phase, the copied actors (which will eventually comprise the daughter) form a reversed chain in the *leaving* state attached to the mother's head.

Decoding

The *tip* graph rewrite rule from behavior H (possessed only by actors of type #) recognizes when the head actor has been copied and begins the decoding phase, implemented by graph rewrite rules from behavior P. In the decoding phase, the reified-string is interpreted as an adjacency list representation of the behavior graph. This is accomplished as the copied actors traverse the mother a second time (in the reverse direction). During this traversal, each actor has its type specific behaviors conferred on it. The *key* rewrite rule causes actors denoting behaviors adjacent to actors of type # to enter the *key* state. Actors in the *key* state *unlock* adjacent actors of matching type in the *locked* state while

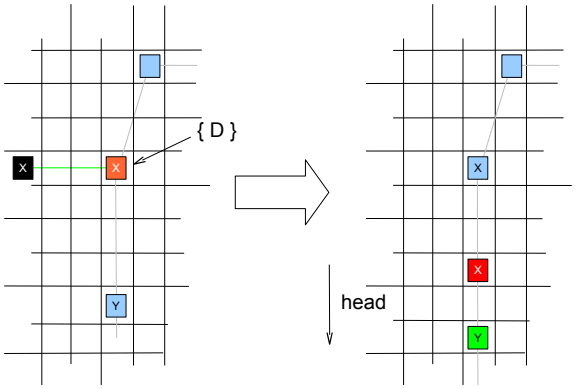

Figure 3: *Insert* graph rewrite rule. An actor in the *inserting* state possessing behavior D waits until its *prev* bond is of maximum length. It then inserts the actor at the end of its hand into the reified-string by bisecting the *prev* bond and enters the *default* state. The inserted actor's state is changed to *leaving* and the state of the actor previously at the end of the *prev* bond (and nearer the head) is changed to *grabbing*.

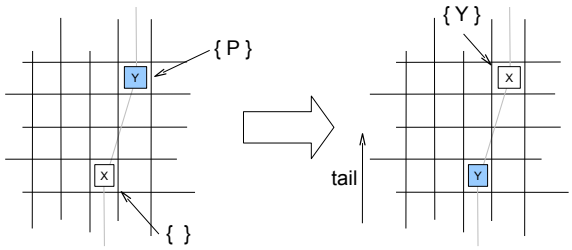

Figure 4: *Confer* graph rewrite rule. An actor in the *default* state possessing behavior P and denoting behavior Y confers behavior Y on the actor nearer the head when that actor is in the *unlocked* state. It then exchanges position with it, moving it towards the tail.

actors in the *default* state *confer* the behaviors they denote on adjacent *unlocked* actors (Fig. 4). Finally, actors in both *locked* and *unlocked* states are moved towards the tail.

The daughter's actors, now possessing their full complement of behaviors, are assembled into a complete reified-string at the end of a *hand* bond at the mother's tail by graph rewrite rules from the L behavior. When an actor in the *last* state sees two others denoting the same behavior as itself at the end of its hand, it sets both its own state and that of the nearer of the two to *grabbing* and deletes its hand (Fig. 5). This separates mother and daughter reified-strings and initiates the process of self-replication in each.

Reified-Set Quine

In the reified-string quine, the behavior graph was encoded using an adjacency list representation, which is capable of representing arbitrary graphs. However, the reified-string quine's behavior graph was far from general–two behaviors

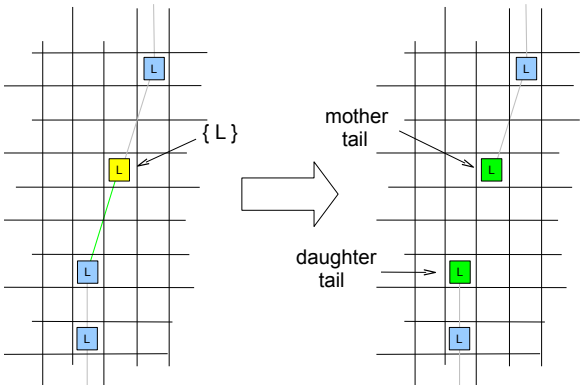

Figure 5: *Cleave* graph rewrite rule. When an actor in the *last* state sees two others denoting the same behavior as itself at the end of its hand, it sets both its own state and that of the nearer of the two to *grabbing* and deletes its hand.

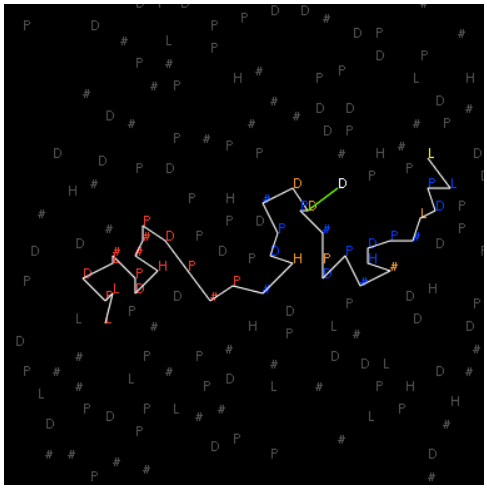

Figure 6: Reified-string quine with *hand* bond (drawn green) in the middle of the *copy* phase. Letters indicate actor type and colors indicate actor state.

(D and P) were possessed by all actors while the remaining behaviors (H and L) were possessed by only a single actor each. If we restrict ourselves to behavior relations comprised solely of *generic* behaviors and *specialized* behaviors, a more compact encoding scheme can be used.

A *reified-set* is a KA consisting of a ring of reified actors linked by *prev* and *next* bonds. A reified-set self-description which is also a programmable constructor for the class of reified-sets is a *reified-set quine*.

Reified-set quines have one great advantage when compared to reified-string quines, namely, the order of the actors in the ring is unimportant. More precisely, there is an equivalence class of bond graphs which encode a given behavior relation. Because actors can swap positions without changing the encoded behavior relation, they can possess a behavior which continually mixes their positions in the ring, ensuring that any two actors will eventually be adjacent. This

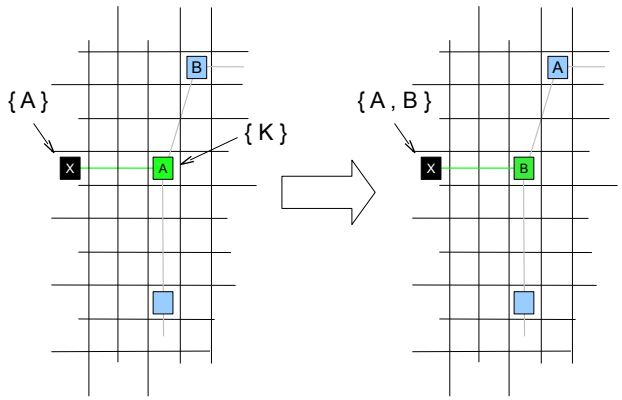

Figure 7: *Shape* graph rewrite rule. An actor in the *shaping* state possessing behavior K, when adjacent to its *continuation*, confers the behavior denoted by its continuation on the actor at the end of its hand. It then exchanges position with its continuation and leaves it in the *shaping* state.

permits a much more expressive form of parallel distributed computation than was possible with the reified-string. Indeed, if each actor in the reified-set possesses a unique *address* and a unique *continuation* then the reified-set can execute sequential programs which perform one operation for every actor. In our work, an actor's address is just the name of the behavior it denotes and its continuation is the name of another behavior. In effect, the reified-set, implemented within a *reified* actor model using *relative* addressing, can simulate a conventional *non-reified* actor model with a small *absolute* address space.

Behaviors

Let X denote a generic behavior and $\underline{X}$ denote a specialized behavior then $Z = \{\underline{C}, K, U, S, N, R, M, \underline{Z}\}$ is a reified-set quine with the following behaviors:

$\underline{C}$ – *create* daughter pinch
K – *find* matching actor, confer type specific behaviors using *shape*, then *splice* it into the reified-set
U – *seek* continuation
S – *swap* positions with adjacent actor
N – *nothing*
R – *ratchet* actors past *pinch* bonds
M – *minimize* bending energy (Williams and Shah, 1992)
$\underline{Z}$ – *fission*.

While the reified-string quine copied itself in two consecutive phases, the reified-set quine copies itself using processes called *copy-decode*, *export*, and *verify* running concurrently in mother and daughter *subrings*.

Copy-decode

Copy-decode is implemented by graph rewrite rules from behaviors K and U. It is a sequential program of sixty four

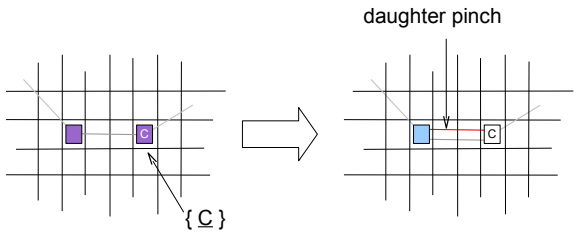

Figure 8: *Create* graph rewrite rule. An actor possessing and denoting behavior $\underline{C}$, in the *going* state, when adjacent to another actor in the same state, forms a *pinch* bond with the adjacent actor and enters the *checking* state (initiating the *verify* program in the daughter subring). The state of the adjacent actor is set to *ready*.

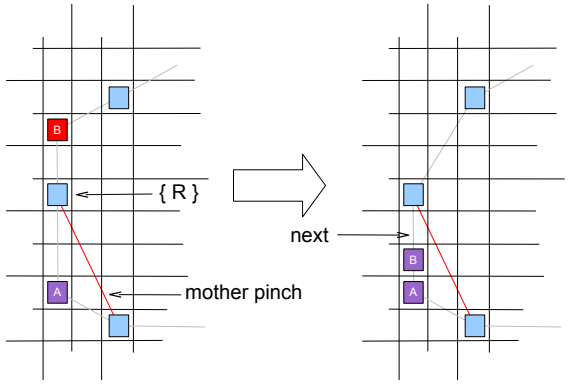

Figure 9: *Ratchet* graph rewrite rule. When at the front end of the mother's *pinch* bond, and adjacent to an actor in the *going* state, an actor with behavior R waits until its *next* bond is of maximum length. It then moves the adjacent actor past the *pinch* bond by bisecting the *next* bond, leaving the exported actor in the *gone* state.

steps which runs in the mother subring and is comprised of two nested loops–the outer loop copies the bond graph and the inner loop decodes the set representation of the behavior graph. Both loops iterate over the eight actors in the reified-set. The outer loop begins when an actor in the *finding* state:

- forms a *hand* bond to an unbonded actor of matching type in its $n \times n$ neighborhood
- gives the daughter actor the name of its continuation (it will be the name of the daughter actor's also)
- enters the *shaping* state.

An actor in the *shaping* state waits for its continuation to be adjacent. When this happens, the actor:

- confers the behavior denoted by its continuation on the daughter actor at the end of its hand (Fig. 7)
- swaps positions with its continuation (leaving it in the *shaping* state)
- enters the *pending* state.

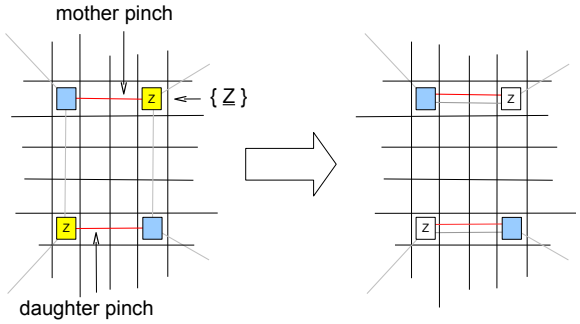

Figure 10: *Fission* graph rewrite rule. An actor possessing and denoting the behavior $\underline{Z}$, when in the *fission* state and located at either end of a *pinch* bond, looks for a second actor in the *fission* state in its bond graph k neighborhood in the other subring. If one exists, the *prev* and *next* bonds joining mother and daughter are rerouted so they coincide with the mother and daughter *pinch* bonds; the actors in the *fission* state enter the *seeking* state.

This begins the next iteration of the inner loop. The inner loop continues until an actor in the *shaping* state finds itself adjacent to its *pending* continuation. When this happens, the inner loop exits and the actor enters the *splicing* state. An actor in the *splicing* state waits until its *next* bond is of maximum length. When this happens, the actor:

- inserts the actor at the end of its hand into the reified-set by bisecting the *next* bond (leaving it in the *going* state)
- enters the *seeking* state.

This begins the next iteration of the outer loop. When an actor in the *seeking* state possessing and denoting behavior $\underline{Z}$ finds itself adjacent to its *pending* continuation the copy program has finished, and the actor enters the *fission* state. It remains in the *fission* state until the *verify* process (running in the daughter subring) also completes.

Export

Export is implemented by a set of graph rewrite rules from behaviors S, R and $\underline{C}$ which run concurrently with *copy-decode* and *verify* in both mother and daughter subrings. The *swap* graph rewrite rule swaps actors in the *ready* state with actors in posterior positions; a second rule portages actors around actors with *hand* bonds. These rules serve two purposes. First, they continually mix the positions of the actors in both the mother and daughter subrings, ensuring that any two actors in the same subring will eventually be adjacent. This is necessary for the *copy-decode* and *verify* programs to make progress. Second, they cause actors in the *going* state in the mother subring to move towards the gate formed by the mother and daughter *pinch* bonds–bonds created by the single rewrite rule in behavior $\underline{C}$ (Fig. 8).

Graph rewrite rules from behavior R control a gate formed by a pair of parallel *pinch* bonds which separate the mother and daughter subrings. Another graph rewrite rule swaps pairs of actors joined by *pinch* bonds. This routes actors in the subrings across the pinches, effectively short-circuiting the mother and daughter subrings and ensuring that the mother's and daughter's actors cannot mix. Indeed, the only actors which can get past the mother's *pinch* bond are actors in the *going* state and they can only pass in one direction. The actor at the front end of the mother's pinch bond, when adjacent to an actor in the *going* state, waits until its *next* bond is of maximum length. It then moves the adjacent actor past the *pinch* bond by bisecting the *next* bond, leaving the exported actor in the *gone* state (Fig 9). Another graph rewrite rule performs a similar operation at the back end of the daughter's *pinch* bond, leaving the imported actor in the *ready* state.

Verify

Verify ensures that the daughter has received the full complement of actors before fission occurs. It is implemented by graph rewrite rules grouped in behaviors U and $\underline{Z}$. One might think that fission could occur as soon as the actor which is copied last is imported into the daughter subring. However, because of the asynchronous nature of the *export* process, there is no guarantee that the last actor copied will be the last one imported. In fact, import order inversions are common. For this reason, a simple eight step program (one for each actor in the reified-set) is run in the daughter subring to verify that the full complement has been imported.

An actor in the *checking* state in the daughter subring waits until it finds itself adjacent to its continuation. When this happens, it enters the *ready* state and sets the state of its continuation to *checking*. The one exception is the actor representing the behavior $\underline{Z}$–this actor is copied last and does not seek its continuation but enters the *fission* state instead.

An actor possessing the behavior $\underline{Z}$, when in the *fission* state and located at either end of a *pinch* bond, looks for a second actor in the *fission* state in its bond graph k neighborhood in the other subring. If one exists, the *prev* and *next* bonds joining mother and daughter are rerouted so that they overlap the *pinch* bonds; the actors in the *fission* state enter the *seeking* state, initiating the *copy-decode* program in mother and daughter, now separate (Fig. 10).

Discussion

In the introduction, an analogy was made between enzymes and actors, and it was suggested that the primary computational function of a cell's membrane is to create an address space within which actors can send and receive messages without interference from the actors of other cells. The analogy is compelling. However, we have deliberately avoided calling the movable feast an *artificial chemistry*. One reason for not doing so is that we are trying to achieve with *dozens* of actors what is accomplished in a biological cell by *billions* of enzymes. If we are to succeed then we cannot be too literal in our imitation of the biological cell; our goal

should be to build an airplane not a bird.

Communication

Hutton (2007) states that the primary obstacle to constructing an artificial cell with a complete set of enzymes of the sort he has described is the unwieldiness of the vastly larger genome and membrane such a cell would require. However, a more fundamental obstacle may be the difficulty of ensuring communication between enzymes and locations where reactions need to be catalyzed.

Do the enzymes of an artificial cell need to be confined within a 2D space bounded by a 1D membrane? Or can they comprise the membrane itself? Both approaches isolate a cell's enzymes from those of other cells. The second has the advantage that a simple mixing behavior guarantees communication between enzymes and locations where reactions need to be catalyzed.

Modularity

All quines are grounded in terms defined externally in the host programming language. A programming language can have terms which are elementary and general (like Lego bricks) or complex and highly specialized (like stereo components). The terms can have uniform interfaces (like USB devices) or interfaces which limit reuse (like the pieces of a jigsaw puzzle).

The terms comprising the genome of the reified-set quine are behaviors defined outside the quine itself. A crude upper bound on the number of reifed-set quine genomes would be 2^B where B is the number of behaviors. Of course B can be made arbitarily large initially, but wholly new behaviors cannot evolve; evolution is limited to discovering viable combinations of pre-existing behaviors.

Do these exist? Are there viable and interestingly different reified-set quines near Z in genome space? In partial answer to this question, we have constructed two additional examples of reified-set quines which use very different strategies to ensure that the daughter cell has received its full complement of actors. The first, X, accomplishes this by running a second instance of *copy-decode* inside the daughter subring instead of *verify*. In effect, the daughter demonstrates its viability by constructing the granddaughter. The second, Y, uses a modified *copy-decode* program which waits until it sees the most recently copied actor in the daughter subring (through the pinches) before it continues.

All three reified-set quines share behaviors K, S, R and M while two (X and Z) also share U. This demonstrates that behaviors can possess a degree of modularity and potential for reuse and can be mixed and matched meaningfully. While the three reified-set quines were designed and did not evolve, the fact that they exist suggests that a future system more like Hutton (2007), with a genome containing reified descriptions of graph rewrite rules subject to mutation, would explore a genome landscape populated by viable and interestingly different artificial cells.

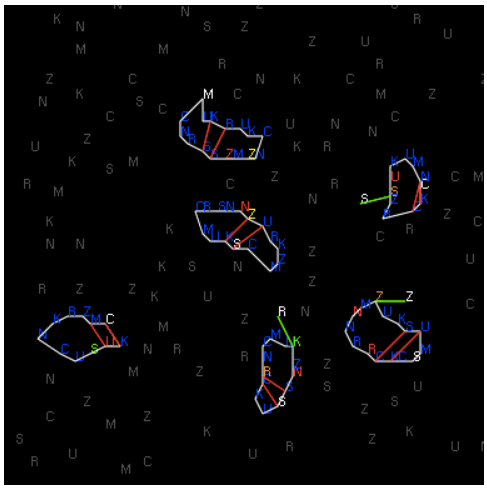

Figure 11: Six reified-set quines. Letters indicate actor type and colors indicate actor state. In the mother subring of the topmost quine, the *copy-decode* program has completed, while the *verify* program is still running in the daughter subring. *Hand* and *pinch* bonds are drawn green and red.

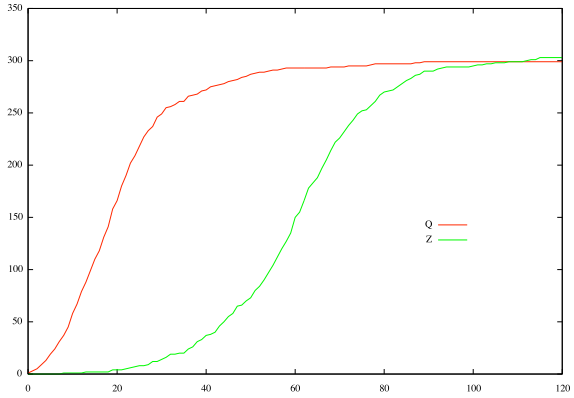

Figure 12: Exponential growth of non-competing populations of reified-string quines, Q, and reified-set quines, Z.

Experimental Results

In each of the three experiments, approximately 11000 unbonded actors were randomly placed on a grid of size 512×512 to achieve a 4% area density. Except for pairs joined by *prev* or *next* bonds, actors were excluded from 5×5 neighborhoods surrounding other actors. The maximum bond length equaled 4, the diffusion constant equaled 0.5, and search neighborhoods were of size 11×11.

In the first experiment, the unbonded actors were of types comprising the genomes of the Q reified-string quine and the Z reified-set quine. The proportion of each type matched that of the two genomes. A single reified-string quine and a single reified-set quine were then placed in the grid, after which, populations of both increased exponentially, in the

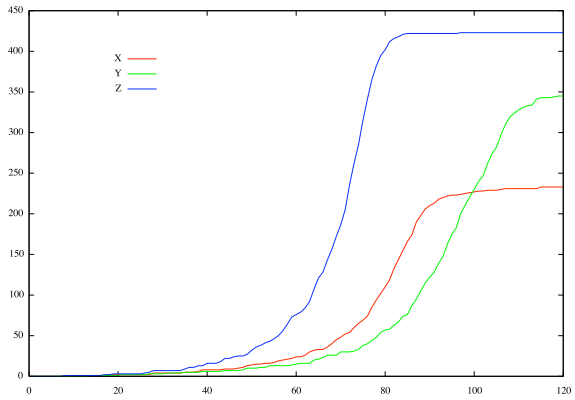

Figure 13: Exponential growth of three non-competing populations of reified-set quines. Z is the quine described at length in this paper while X and Y use alternative strategies to verify that the daughter has received its full complement of actors.

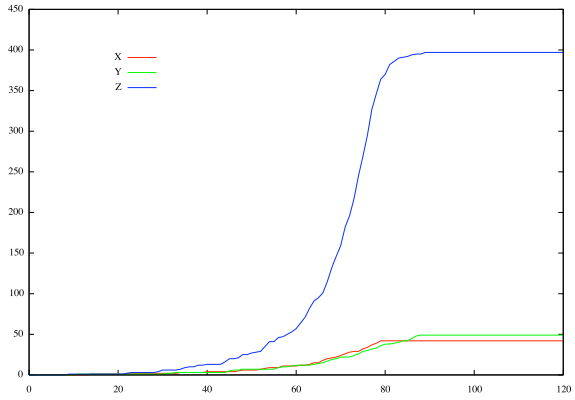

Figure 14: Three populations of reified-set quines compete for a shared resource, the K behavior. The Z quine outcompetes the X and Y quines.

process converting essentially all unbonded actors into approximately 300 copies of each quine (Fig. 12).

In the second experiment, the unbonded actors were of types comprising the genomes of the X, Y, and Z reified-set quines. As before, the proportions of each type matched those of the genomes; types common to all three, *e.g.,* K, were three times as numerous as unique types, *e.g.,* U. Single X, Y, Z reified-set quines were then placed in the grid. Populations of all three increased exponentially, yielding 424 copies of the Z quine, 345 copies of the Y quine, and 233 copies of the X quine (Fig. 13). The differences in these numbers can be attributed to the fact that (after all unbonded actors have been consumed) the final population consists of a mixture of individuals at various points in the self-replication process and which therefore exhibit a range of sizes. The Z quine is the most efficient at converting unbonded actors into copies of itself while the X quine is the least. This is presumably due to the fact that the X quine requires its daughters to demonstrate their viability by constructing grandaughters. Consequently, the average size of X quine intances is significantly larger than the average size of Y or Z quine instances.

The conditions of the third experiment were nearly identical to those of the second except that the number of unbonded actors of type K (common to all three genomes), was reduced by a factor of three. Consequently, populations of X, Y, and Z quines were forced to compete for the underrepresented shared resource. The winner of the competition was the Z quine, which succeeded in constructing nearly 400 complete individuals, while the X and Y quines succeeeded in constructing less than 50 each (Fig. 14).

Conclusion

A highly expressive, indefinitely scalable, and asynchronous model of parallel distributed spatial computation has been introduced and used to define a series of self-replicating kinematic automata. These machines assemble copies of themselves from components supplied by diffusion and increase in number exponentially until the supply of components is depleted. Because they are both programmable constructors and self-descriptions, we call them *reified quines*.

Acknowledgements

Thanks to Dave Ackley for many helpful conversations.

References

Ackley, D. H. and Cannon, D. C. (2011). Pursue robust indefinite scalability. In *Proc. HotOS XIII*, Napa Valley, CA, USA.

Brand, D. and Zafiropulo, P. (1983). On communicating finite-state machines. *J. ACM*, 30:323–342.

Burks, A. (1970). von Neumann's self-reproducing automata. In Burks, A., editor, *Essays on Cellular Automata*, pages 3–64. University of Illinois Press.

Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245.

Hutton, T. J. (2005). Replicators that make all their own rules. In *Proc. Workshop on Artif. Chem. and Its App., 8th European Conf. on Artif. Life*.

Hutton, T. J. (2007). Evolvable self-reproducing cells in a two-dimensional artificial chemistry. *Artif. Life*, 13:11–30.

Klavins, E., Ghrist, R., and Lipsky, D. (2004). Graph grammars for self assembling robotic systems. In *IEEE Conf. on Robotics and Automation*.

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

Turing, A. (1936). On computable numbers, with an application to the Entscheidung problem. *Proc. London Math. Soc.*, 2(42):230–265.

Williams, D. and Shah, M. (1992). A fast algorithm for active contours and curvature estimation. *CVGIP: Image Understanding*, 55(1):14–26.