MetaChem: An Algebraic Framework for Artificial Chemistries

Penelope Faulkner Rainford
University of York
Department of Chemistry
York Cross-disciplinary Centre for Systems Analysis
University of Hull
Business School
p.rainford@hull.ac.uk

Angelika Sebald
University of York
Department of Chemistry
York Cross-disciplinary Centre for Systems Analysis
angelika.sebald@york.ac.uk

Susan Stepney*
University of York
Department of Computer Science
York Cross-disciplinary Centre for Systems Analysis
susan.stepney@york.ac.uk

Abstract We introduce MetaChem, a language for representing and implementing artificial chemistries. We motivate the need for modularization and standardization in representation of artificial chemistries. We describe a mathematical formalism for Static Graph MetaChem, a static-graph-based system. MetaChem supports different levels of description, and has a formal description; we illustrate these using StringCatChem, a toy artificial chemistry. We describe two existing artificial chemistries—Jordan Algebra AChem and Swarm Chemistry—in MetaChem, and demonstrate how they can be combined in several different configurations by using a MetaChem environmental link. MetaChem provides a route to standardization, reuse, and composition of artificial chemistries and their tools.

Keywords
Artificial chemistry, swarm chemistry, nested chemistry, Jordan algebra, MetaChem, algebraic framework

1 Introduction

The field of artificial chemistry covers many rich and diverse systems [1]. Yet practitioners can struggle to talk about specific systems in the context of the whole field. It is hard to make comparisons between even intuitively quite similar systems. For example, consider a toy AChem that concatenates strings of letters from the Roman alphabet; now consider another toy AChem that combines lists of

* Corresponding author.

integers with values between 1 and 26. There is a clear mapping between the two sets of particles, and they use equivalent linking rules. However, such similarities can be hidden, for example by describing one as a string system and the other as a list system. Then, despite the underlying equivalence of the particles and linking rules, their overall behavior might be very different if, say, strings can decompose but lists cannot, or if, say, lists move in simulated space and use collisions to determine linking, while strings interact in a well-mixed tank. Are they fundamentally equivalent, or fundamentally different, and where do the differences lie? In other cases, systems may have similar goals but very different components, algorithms, and implementations. This makes building even a basic classification system challenging.

To tackle this issue, Dittrich et al. [2] declare that an artificial chemistry (AChem) is described by the triplet (S, R, A): a set of particles S, rules for reactions R, and the algorithm A. The particles and reaction rules are reasonably clearly circumscribed, but all the other aspects of the system are combined in A as part of the algorithm, covering such diverse concepts as spatiality, rule application, environmental conditions, timing, and logging.

Despite its apparent generality, the (S, R, A) model does not accommodate all systems that practitioners may want to regard as AChems. A different conceptual view of artificial chemistry may be more inclusive of all AChem systems. Consider: An artificial chemistry is a system with minimal components designed to use or explore the higher-order emergent properties of their interactions. This conceptualization allows for AChems with purely kinetic interactions, and for ones that do not distinguish between the R and A aspects. It also acknowledges that some systems are distinguished as different due to the differences in A rather than S or R.

In the (S, R, A) model, many features of more recent chemistries are lumped into A. This one component contains huge amounts of important parts of an AChem, bundled as "everything else." Aspects of a system that may indicate intent, or overdesign towards a goal, can be lost in A.

Additionally, it is not always clear how to partition the design of an AChem into these three components. If a system has to use an extra reaction to make membranes possible, where do we find this in (S, R, A)? If a system is probabilistic on account of a property of the environment, is that in R for the reaction, or part of A the algorithm? If different designers make different decisions for the same feature, it is hard to compare their systems. For example, there are now three different versions of random-Boolean-network-based subsymbolic AChems [4, 12, 32], and subversions within each version. These all have essentially the same underlying S, but subtly different bonding rules R, and a variety of very different A used to explore different properties. Are the different results due to the changes in R or in A? Overall, it is impossible to rigorously define and quantify differences or similarities between systems with such a crude language of only three words.

The (S, R, A) description was a good tool when it was developed, when AChems were often still small toy systems, or even just thought experiments. It was sufficient for the many early "proof of concept" AChems that demonstrate that an artificial system can produce cell-like objects capable of self-maintenance and self-replication. However, as AChems move into new realms, being used for computation and as the basis for open-ended evolution systems, we need to be able to analyze them more rigorously, make comparisons between alternative models, and reuse components between models. We need to develop a new, more sophisticated descriptive framework for artificial chemistries.

Here we present MetaChem, a formal description language for AChems, which allows us to model, standardize, compare, and combine diverse AChem systems. The structure of the article is as follows. In Section 2 we discuss the objects and actions common to AChems, to motivate our design. In Section 3 we introduce the basic graph structure of MetaChem, with the various node and edge types that can be used to construct an AChem definition. In Section 4 we demonstrate how this graph structure can be used to define an AChem at different levels of description. In Section 5 we describe the internal structure of the executable nodes, showing how the graph forms a program to execute an AChem; a mathematical definition is provided in the Appendix. In Sections 6 and 7 we recast two widely differing AChems from the literature into MetaChem, to demonstrate its breadth of applica-

bility. In Section 8 we combine these two AChems to produce a family of nested AChems, in order to demonstrate the combinatorial power of MetaChem. We conclude with a discussion of future extensions to MetaChem to allow dynamically changing graphs.

2 Properties of Artificial Chemistries

There are axiomatic concepts that we build on in the field of artificial chemistries. AChems start with small components interacting to generate our systems. Analysis tends to focus on the emergent properties and behaviors of these systems. To differentiate an AChem from an individual-based model [8], we add requirements for simplicity (ease of description) and tractability (ease of computation) in our particles and their interactions. The intention is that these systems work over large collections of individuals and over long time periods, although most are currently limited by computational capability.

From these concepts we identify many common elements of AChem systems, and use these as the basis for a bottom-up approach to systematic modularization of AChem systems. Individual *particles* and their *interactions* are our primary focus. These are present throughout AChem systems. Systems also have other variables, properties, and values. Much as in real chemistry, we separate the description of the "glassware" from our consideration of its particle contents. We separate these other values and properties into an *environment*. We can have multiple *containers* in our systems, which allow us to isolate particles and move them, analogous to the "beakers," "tubes," and "valves" constituting the "glassware," or membranes and compartments in biological cells. These components constitute the "things" in our systems (Table 1).

There are also commonalities in the algorithms of AChems (and often their implementations) that we abstract out in our framework. Control flows, related to time and generations, occur in most systems: Some systems update across all objects in the system at once; others continuously update objects at random. If we can identify the modularized control that produces these timing systems, designers could switch between them. This would then allow designers to focus on the new AChem-specific features of their design, while exploiting preexisting elements to implement less unique aspects of their systems.

We define our control flow in relation to how we divide our *things*. We *modify particles*, similarly to reactions and interactions in chemistry. We *record observations* of our system. We *modify our environment*, such as by changing the temperature of the system. We *move* particles around our system. We *decide* which of these things we should do next. These control flow actions form the building blocks of our MetaChem.

3 Modularization: Components of an Artificial Chemistry System

We arrange these concepts into the structure shown in Figure 1, which we use to build a graph-based formalism. We have the overarching concepts of the system, made up of the elements formalized as graph *nodes* (containers, control), and as graph *edges* (information flow, control flow).

Table 1. Common parts of artificial chemistry systems.

	Primary focus	Auxiliaries
Objects	Particles	Variables
Containers	Tanks	Environment

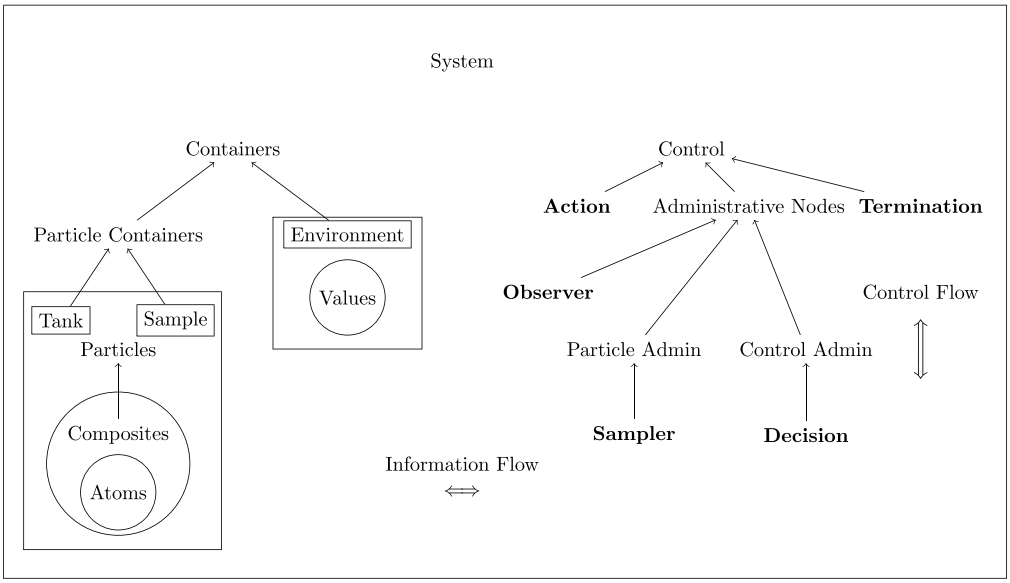

Figure 1. Conceptual structure of modularization of artificial chemistries.

Control items are static nodes in the graph: Their locations and connectivity are defined at the start of a *run*, and remain unchanged as the AChem executes. (Future versions will support dynamic connectivity; see Section 9.2.3.) These control nodes are connected by control flow edges, which together define the system's algorithm.

Containers are also static nodes in the graph. They map to (*contain*) the dynamically changing particles and environmental values in the system.

Element	Description
Containers	
T	**Tank**: particle container
S	**Sample**: particle container of editable particles
V	**Environment**: container of non-particle variables and information in the system
Control	
s	**Sampler**: Information administration node that moves particles between containers
o	**Observer**: Information administration node that observes particles in containers, and saves summary statistics into the environment
d	**Decision**: Control administration node that decides on control flow path based on the state of particles and the environment
a	**Action**: Control node that performs actions on particles based on state of particles and environment
●	**Termination**: Control node where processing terminates

Figure 2. Graph node types, and their graphical representation, used in MetaChem. The initial control node (typically a sampler node to load some initial state) is identified with a double border; see Figure 6.

Element	Description
Information Flow	
	**Read**: Allows reading of information from source container node (shown) into target control node local state.
	Pull: Allows pulling of information out of source container node (shown) by target control node. Also allows reading from the source container.
	Push: Allows pushing (writing) of information from source control node local state into target container node (shown). Also allows reading from the target container.
Control Flow	
	Solid arrow between control nodes indicates control flow in system.

Figure 3. Graph edge types, and their graphical representation, used in MetaChem.

Information flow edges allow control to influence the connected containers' states (that is, contents). Information can flow in either direction along an edge: *read* or *pulled* from containers' states to the control node, and *pushed* from the control node to update containers' states.

These general concepts are captured by different types of graph nodes (Figure 2), and types of graph edges (Figure 3). We can use these to define graphs that provide a view of our systems. We can provide different views using graphs at different levels: from a high-level overview, down, by expanding nodes, to levels with greater detail (Section 4).

3.1 Particles

The most fundamental parts of our systems are the *particles*. These and their emergent properties and behaviors are the focus of our studies. These can usually be split into two subsets: *atoms* and *composites*. Some AChems may have only atoms and, due to lack of physical bonding rules, may not seem to form composites. Others may be symbolic and assume that all particles are complex and that all the symbols represent composites.

Atomic particles: the most basic particles; they cannot be divided or broken down into smaller parts. Any internal structure of the atoms [5] is indivisible.

Examples. Atoms can take many forms: characters in a string chemistry, instructions in an automata chemistry, or symbols that are not the "one" in a one-to-many symbolic production rule in a symbolic chemistry.

Composite particles: these are are made of combinations of atoms. In symbolic AChems the atoms making up a composite particle may be hidden or unknown.

Examples. Composites would be strings in string chemistries, programs in automata chemistries, or symbols that result from many-to-one rules in symbolic chemistries.

3.2 Container Nodes

Container nodes are partitioned into two subtypes: particle container nodes and environment container nodes.

Particle container nodes: mappings that take the node and the state of the system, and return the multiset of particles in that container at that state. When the system is in a particular state, the set of mappings of all the containers forms a partition of all the particles in the system.

There are two types of particle container nodes: *samples* and *tanks*. Tanks are protected containers, in that particles inside them cannot be edited. Particles in tanks can be moved in and out, but cannot be changed; any changes must be made over samples, so that the designer must explicitly decide what will be changing.

Examples. A beaker being used for an experiment; a pipette; a petri dish.

Environment container nodes: similar to particle container nodes, except that they contain non-particle objects and information in the system. The system can have multiple environments, to make reference to the things in the environment easier. For example, one might want to store a time record separately from summary statistics or log information; one might want different local temperatures for different containers. These are all accessed via some mapping from the node and state of the system to the dynamic information and objects.
Examples. Temperature readings; Bunsen burner; stirrer; observation log.

Container nodes are never directly connected to each other. All communication between them is mediated by control nodes. This means we always have control over the movement, similar to having valves and drip taps installed in ordinary chemistry glassware. We can allow things to flow through these controls freely, but we always have the option to restrict or stop any flows.

3.3 Action Nodes

Action nodes, a kind of control node, are where we actually modify particles through movement, linking, decomposition, or any other change. Actions can modify particles only in a *sample*. This means we always need to designate which particles we are changing before change occurs. This protects the particles in *tanks*.
Examples. Concatenate strings; form chemical bond; execute an automata chemistry program string.

3.4 Admin Nodes: Sampler, Observer, Decision

Admin nodes, kinds of control nodes, are where particles and environments are moved and inspected.
Sampler: Information Admin nodes that move particles between particle containers (*tanks* and *samples*).
Examples. Extracting a sample with a pipette for testing; choosing a neighbor to combine with the current particle.
Observer: Information Admin nodes that observe particles and/or the environment state of other nodes. They do not change any internal properties of particles or move them between containers. They can only see particles, derive information such as summary statistics, and modify the environment.
Examples. Taking notes in a log book; updating time in a discrete-time system; updating the number of particles in the system.
Decision: Control Admin nodes used to change control flow. This is the only place control flow can branch, by using information the node has read from connected containers.
Examples. Triggering an event; continuing to the next phase; looping over a process; completing a time step; deciding to take a beaker off the heat.

3.5 Termination Node

Termination nodes, a kind of control node, are where execution of the AChem is explicitly terminated. For an executable system, this implies the need for non-volatile memory for at least some containers, so that their contents can be inspected after a run; this is an implementation issue.

Not all graphs need to have an explicit termination node: We can define an *open-ended* AChem that implicitly runs forever.

3.6 Edges

The nodes of our graphs are connected by edges capturing two kinds of relationship (Figure 3).
Information flow: This first relationship marks movement of information between nodes. Such relationships are always between container nodes and control nodes. Container nodes cannot directly transfer information; the same is true of control nodes.

Figure 4. A combination of a read, a push, and a pull edge, abbreviated as a double-headed information edge.

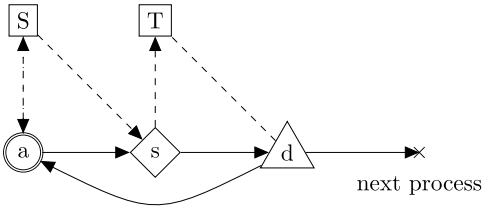

Figure 5. Example of a graph depicting actions and decisions. Control starts in the action node **a** (the double-edged icon indicates it is the initial node in the system), which can pull and push particles from and to the sampler node **S**. Next, control moves to the sampler node **s**, which can pull particles from the sampler node **S**, and push particles to the tank node **T**. Next, control moves to the decision node **d**, which has two control edges coming out of it. Based on the working of the decision node, using information it reads from the tank node **T**, control either loops back to the action node **a**, or continue on to the next process in the system.

Example. Figure 4 shows that an action node reads information from a sample, pulls a subset out of the sample (such as removing two particles to combine), performs its processing (such as combining the two particles), and then pushes (writes) the results back into the sample.

Control flow: This second relationship type marks the movement of control. The edges are between control nodes, and indicate in what order we visit the control nodes. For most control nodes there can be only one outgoing control edge. The exception is decision nodes, whose purpose is to provide a branch point in control flow.

Example. Figure 5 shows the use of a decision node to control looping. An action node links particles in the sample (as in the example shown in Figure 4). Next the sampler node moves the content of the sample container to the tank. Then the decision node checks if the system is finished with linking; if not, control loops back to the action node, to continue linking, otherwise control continues on to the next process in the system.

3.7 Graph as an Executable Algorithm

We have so far discussed separating out the parts of an AChem into nodes and forming a graph using information and control flow edges between these nodes.

This graph is an executable script; it can be executed in software [19]. During execution we have a single execution pointer[1] on a control node that executes a transition function, and then follows the control edges, moving around the graph and executing the transitions defined in the control nodes (see Section 5 for how the internal processes of nodes are defined).

In this version a graph is a static object that is defined before execution and remains unmodified by execution. This is the initial static graph form of MetaChem.

Information flow edges can be seen as directing input and output of the nodes. In a way container nodes act like "blackboard systems" [9], being constantly modified and updated by "experts," the control nodes. Samples exist to section off part of a container and thereby to control which parts of our "blackboard" each of our "experts" can edit. In terms of a physical blackboard, they allow us to draw a box around the content of our tank and write "Do not erase!" next to it.

1 Later versions might support multiple threads of control.

4 Descriptive Levels

The MetaChem graphical formalism allows a modular description of an AChem in terms of its subcomponents. The level at which we define these subcomponents gives the *descriptive level* of our graph.

4.1 Expanding and Summarizing

Moving between descriptive levels may expand nodes into subgraphs, or summarize subgraphs as nodes.

In the case of expansion, the resulting subgraph can still be described in terms of the component functions of the original single node. Such a graph can therefore be summarized in a well-defined manner as a single node.

Starting with an arbitrary subgraph and summarizing it into a single node is in general harder. If the subgraph we wish to describe as a single node can be broken up into the different component functions, then we can summarize it as a node with these functions. Failing this, we summarize as follows:

- All information needed in the subgraph is read in during the node's read phase, so the new node has all the read connections to containers in the larger graph that exist in the subgraph.
- Any samples and variables are also taken; in some cases where there is sampling from a tank, this will need to occur in a separate sampler node. This will give all the container connections needed.
- We then perform all processing in the subgraph in a single action node, including observations.
- Pushing samples to tanks requires another separate sampler node.

So in the worst case we can summarize any subgraph as at most three nodes.

4.2 Node Names

With these different levels and complex systems and multiple nodes of the same type, the basic single-letter tags used before are not sufficient. In order to distinguish the different nodes, we use tags with two-part names.

For containers we write **X:**label, where $\mathbf{X} \in \{\mathbf{T}, \mathbf{S}, \mathbf{V}\}$, and for control nodes we write **x:**label, where $\mathbf{x} \in \{\mathbf{s}, \mathbf{o}, \mathbf{d}, \mathbf{a}, \mathbf{t}\}$. The type tag, **X** or **x**, is part of the overall name.[2] The label is a nonempty string of alphanumeric characters and underscores.

We can use the same label for different types of containers, for example, **T:**particles and **S:**particles for a tank and a sample container of particles, or **S:**sample and **s:**sample for a sample container and its control node. In the first example we are labeling based on the sort of content, which is the same for both the tank and the sample. In the second we label the function in the system, the container contains a sample, and the sampler takes a sample. These labels can be used in the same graphs for different nodes, as the type is part of the node name.

Container nodes with the same name in a graph represent the same node: They may be drawn separately for clarity.

Control nodes with the same name are not necessarily the same node, but do apply the same process to the data they read in: They have identical internal functions. However, they read in from different containers given by the information edges, and transition to a different node after they are completed, given by the control edge. Since they have no memory, control nodes with the same

2 A form of "Hungarian" notation.

name and the same information and outgoing control edges are equivalent nodes, and could be replaced with a single node. Since the node receives no information directly from the previous control node, and can have multiple incoming control edges, these edges are not important for node equivalence.

4.3 StringCatChem: An Illustrative Toy Example

To illustrate the use and power of MetaChem at different descriptive levels, we introduce StringCatChem, a toy AChem. StringCatChem is simple and small enough to run by hand, and can be fully and succinctly decomposed into its parts. We use MetaChem to describe full-scale AChems in later sections.

In StringCatChem the atoms are character strings, and composites are formed by string concatenation. StringCatChem is situated in a collection of well-mixed tanks. When a string is selected for reaction, it is checked whether it contains any identical adjacent letters; if so, it is split between them. If not, a second string is selected at random from the same tank, and the strings are concatenated. The split or combined strings remain in the same tank. In a separate operation, strings are also randomly transferred between tanks.

StringCatChem is very simple, just forming random strings with no double letters. It will continue to act until all the strings have matching letters at the starts and ends, or there is one large string. After this the system will not change, as any concatenation will be split apart again before another concatenation can occur. StringCatChem is therefore not a good choice of AChem if one wishes to study interesting behaviors such as replication, open-endedness, or the transition to life. However, it makes a good illustration of MetaChem: The whole system can be implemented with four container nodes and 13 control nodes.

4.4 Macro Level

The *macro*-level view provides an overview of the entire AChem. It rarely deals with individual values, atoms, or interactions in the AChem. Even a generation or time step at this level is just an update process.

Figure 6 shows the macro-level description of StringCatChem. (A modification for explicit termination, shown in Figure 7, is discussed below.) The start node is the control node with a double border. In many systems—for example, ones that start with an initial set of particles to be loaded into the system, as here—the start node will have no incoming control flow edges.

- StringCatChem starts with the **s**:load node loading a set of strings into the set of tanks.
- The observer **o**:time then increments the time variable.
- The action **a**:process is responsible for the splitting and concatenation reactions that occur in the individual tanks. This is expanded later in Figure 8.

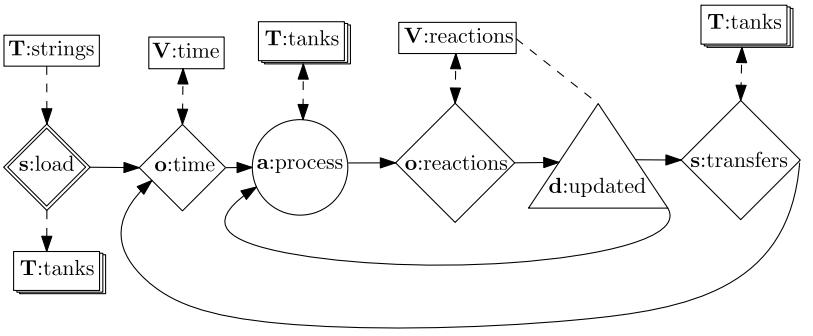

Figure 6. Macro-level description of (open-ended) StringCatChem.

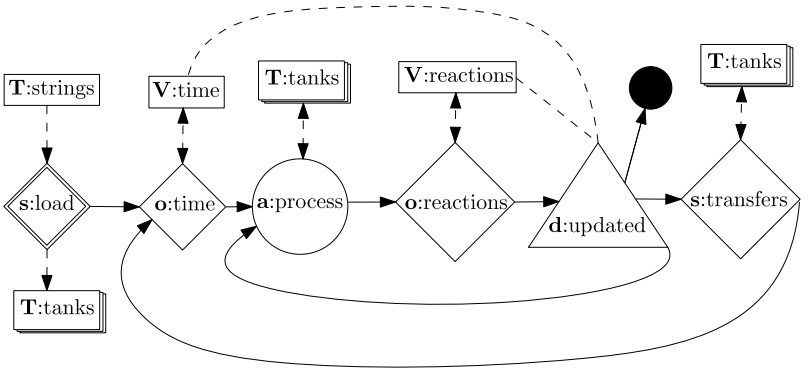

Figure 7. Macro-level description of explicitly terminating StringCatChem.

- The observer **o**:reactions then increments the reaction variable, to keep track of the number of reactions done in this time step.
- The decision **d**:updated checks if the update cycle is complete (if enough reactions have been performed). If not complete, it moves control back to **a**:process. If complete, it moves control on to **s**:transfers.
- The sampler **s**:transfers moves particles between tanks, then loops back to **o**:time for the next time step.

This description gives us a high-level overview of the main operating loops of the system and of the set of significant processes. We can see that there is a random unsynchronized update. Timing is discrete, and there are multiple tanks with movement between them. These are the kinds of elements and properties of a system that should be visible at the macro level.

This is the macro-level description of the system; it is an open-ended system with no inherent termination point. For implementation purposes however we might add an explicit **V:**time-dependent termination, as in Figure 7. Adding explicit termination to an open-ended system is usually done by adding a termination node to the decision at the end of the update loop, normally with the decision based on some variable, such as a time or generation variable, or anything else the designer wants to use to trigger termination of the run.

There is also a textual representation form for these graphs, which can be used for defining and executing them. An example of this textual form can be found in [19].

4.5 Micro Level

The *micro*-level view provides a focus on the actual action and effects on different particles and environments in the system. It can be thought of as the algorithm or pseudocode-level description of the AChem.

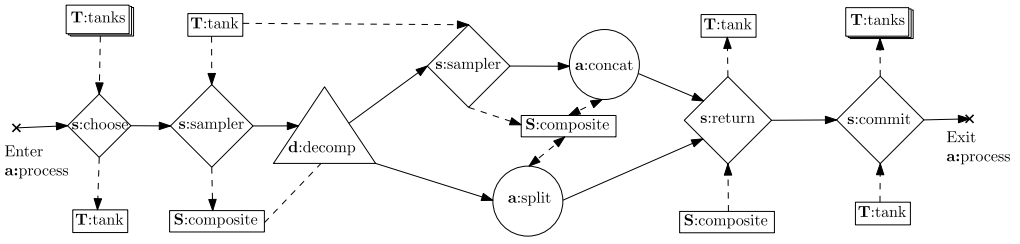

Figure 8. Micro-level description of the **a**:process node of the StringCatChem.

Micro-Level StringCatChem of a:process.
As an example we expand the **a**:process node from Figure 6 into a graph showing the internals of how this action occurs (Figure 8).

In summary, we choose a tank, then choose a particle string from it. We decide (based on the presence of a double character) whether to decompose the string or not. In one case we split the string; in the other we sample a second string and concatenate them. The resulting particle string(s) are then returned to the same tank the original came from, and the tank is returned to the collection of tanks. In terms of the graph, this is described as follows:

- The sampler **s**:choose pulls the contents of a partition from the set **T**:tanks and pushes its contents into **T**:tank.
- **s**:sampler pulls a particle from that **T**:tank and pushes it to **S**:composite.
- **d**:decomp decides if the particle can be decomposed or instead should be concatenated with another particle.
- If the decision is to decompose, control moves to **a**:split, which pulls the particle from **S**:composite, splits it, and pushes the resulting two particles back to **S**:composite.
- If the decision is not to decompose, control moves to **s**:sampler, which pulls another particle from tank **T**:tank and pushes it to **S**:composite; control moves to **a**:concat, which pulls the two particles from **S**:composite, concatenates them, and pushes the resulting particle back to **S**:composite.
- Either path results in control being at **s**:return, which pulls the resultant particle(s) from **S**:composite and pushes them into **T**:tank.
- Finally, **s**:commit pulls the entire contents of **T**:tank and pushes them back into the same partition of **T**:tanks that it originally come from.

4.6 Physics Level

The *physics*-level view deals with the hard-coded details of implementation. It is the designer's choice what is the lowest level of detail needed; anything the designer considers to be unchangeable occurs at this level. This is the full program code-level description, defining the internal processes of the control nodes.

Physics StringCatChem of a:split.
As an example we expand the **a**:split node from Figure 8, which splits a string that contains a double character. In our implementation (available at github.com/faulknerrainford/MetaChem.git), this is defined in the MetaChem package of Python, by subclassing the ControlNode class to provide the specific implementation. The ControlNode class defines the action of any control node in terms of components of the overall transition function, Listing 1 (see Section 5 for details).

The specific node subclass provides an implementation for each of these components in order to define the required processing, Listing 2. The specific implementation is defined as follows:

read() Read the contents of the attached container(s) (`readsample`) into local state. Here `readsample` is **S:**composite, defined by the graph topology on system setup. The local state is `self.sample`.

check() Check that we want to continue processing. Here we use a default check action that returns 0, so the action will always occur. All checks needed have already been made in the previous decision node **d**:decomp (Figure 8), so the action here is deterministic.

Listing 1. Transition function as defined in ControlNode class.

```
1  def transition(self):
2      self.read()
3      if self.check() < random.random():
4          self.pull()
5          self.process()
6          self.push()
7      pass
```

pull() Pull the relevant particles out of the attached container(s). Here the sample should contain only one particle, so we then remove that particle from **S:**composite, using the container's `remove()` function. We specify which particle to remove in terms of the local state of the control node; hence that particle must have been read from the attached container earlier.

process() Process the material in the local state. Here, the process function finds the double letter, and splits the string at that point into two particles or strings, overwriting the internal sample state.

push() Push the relevant particles into the attached container(s). Here we push the two split strings into the `writesample` container, using the container's **add()** function. Here `writesample` is **S:**composite, defined by the graph topology on system setup. The graph topology could later be modified so that `writesample` referred to a different container, without having to change the implementation here.

In the case of most processes the lower-level instructions will use at least some default functionality; in this case it is the check function. In the case of other sorts of control nodes such as samples it might be the process function. These defaults are defined in the relevant superclass code.

All interactions with containers are mediated through the interface of the containers' built-in **read()**, **add()**, and **remove()** functions. This allows the control node design to remain independent of the exact implementation of the containers. With this an AChem designer can build and test a system on the small scale using easy-to-manage list containers, and when they wish to scale up, they can re-implement the containers to use a database, without having to change the graph or the control nodes' implementations.

4.7 Abstraction Levels

The abstraction levels are not restricted to these three levels: There can be systems made up of systems [18] defined using additional levels. It is up to the designer or modifier of the AChem to choose the abstraction level for what is needed and useful in order to properly express and illuminate a particular system.

5 Static Graph MetaChem

Here we describe the internal structures of the nodes, in terms of the actions they perform. We provide a mathematical specification in Appendix A. This is Static Graph MetaChem: None of the actions described here change the structure of the graph. Future versions will include actions that can dynamically change the structure of the graph as the algorithm executes.

Listing 2. **a:**split node as described in Python using StringCatSplitAction. Action is a subclass of the ControlNode class that contains the transition function. All aspects of Action are overwritten in StringCatSplitAction.

```
class StringCatSplitAction(node.Action):

        def __init__(self, writesample, readsample, readcontainers=None):
            super(StringCatSplitAction, self).__init__(writesample, readsample,
            readcontainers)
            self.sample = None
            pass

        def read(self):
            self.sample = self.readsample.read()

        def check(self):
            return super(StringCatSplitAction, self).check()

        def pull(self):
            self.readsample.remove(self.sample)

        def process(self):
            doubleindex =[i for i in range(0, len(self.sample) - 1) if
            self.sample[i] == self.sample[i+1]]
            index = random.choice(doubleindex)
            self.sample =[self.sample[0:index], self.sample[index:0]]
            pass

        def push(self):
            self.writesample.add(self.sample)
```

5.1 Control Nodes and Edges

The control flow defines the AChem's algorithm: Evaluate the current control node's definition, then move to the next control node, and repeat. In the implementation, it executes the current node's transition function, which (potentially) changes state and then moves on to the next node; by traversing the graph in this manner it performs the relevant computation. There is no automatic termination rule on these systems, as chemistries technically don't ever stop, but for a particular algorithm we can define a number of transitions we will perform before stopping. We could also provide a control node with no outgoing edge. This would force termination.

All control nodes have the same basic structure for their state transition function, defined through component transition functions: *read*(), *check*(), *pull*(), *process*(), *push*(), *next*(), executed sequentially:

$$transition = read \ ; \ check \ ; \ pull \ ; \ process \ ; \ push \ ; \ next$$

where ; indicates sequential ordering of function application from left to right.

Each of these component functions plays a different role in the transition and thus uses a different aspect of the state:

read Collect information from (connected) external containers into temporary local containers, for use by the following functions. This action does not modify the external containers; it copies the relevant particles and values into temporary local states.

check Generate a threshold probability value p from local-state information. This p is used to determine if the rest of the component functions (the ones that actually alter containers' contents) occur. In the current implementation, it generates p from its local state, then generates a uniform random number r; if $p < r$, execution continues; otherwise it moves directly to the *next* node. The shape of p can be defined so that the probability of the process follows the desired distribution.

pull Remove particles and change information in external containers. Any information so removed must already have been copied to local containers by the earlier *read* (), where it is available for local processing; such a *read* followed by *pull* has the effect of moving the particles or information. However, read information does not have to be pulled: It can be copied, rather than moved.

process Perform the main computation for the node. This is where the "chemistry" happens. It modifies the state of local particles and variables, including creating new particles and variables and destroying old ones.

push Copy particles and values from local state into external containers.

next Wipe the local state, and move control to the next node. All control nodes except decision nodes have exactly one outgoing control edge, so the move is deterministic. For decision nodes, *process* designates the target node, and stores it in local state. This is used by *next* to move to the chosen next node.

These component functions operate on the node's local state, which exists only for the duration of the overall transition. Local particle containers and local environment containers are destroyed once the transition function is completed, so control nodes have no lasting state or memory. Any information used by a control node must come from containers at the start of a transition by using *read*(); any local information or objects that need to remain in the system must be written back to containers by *push*().

These functions are summarized in Figure 9 and discussed in the context of specific node types below.

5.2 Control-Node Subtypes

Control nodes are partitioned into subtypes: action, decision, sample, observer, termination. We distinguish these node subtypes by requiring some of their transition function components to have no effect (to be the null operation, or the identity transformation), or by limiting the types of containers they can interact with during the transition (Table 2). The constraints on these subnodes help control the complexity of the system definition.

Action: Read in information, check if an interaction occurs, and process the particles in the system for the reaction to happen. This node type is not limited in which transition functions it executes, but it is limited in which containers it can *push*() to (Table 3). The limit to modify only samples allows parallelization, and encourages controlled modification. The designer is required to consider what they wish to modify before they modify it, as they must first *sample* it from the tanks.

Decision: Process the information from its containers and return a choice of the possible next control nodes. It is limited to *read*() and *process*(), so it cannot change the contents of any of the containers.

Sampler: Move particles between containers. It does not compute or process information, and it does not modify any particles or environment variables. It is therefore limited to *read*(), *pull*(), and *push*().

Observer: Observe but do not modify particles; modify the environment. It can *read*() to view containers; it can *pull*() to edit only environment variables. It can *process*() to

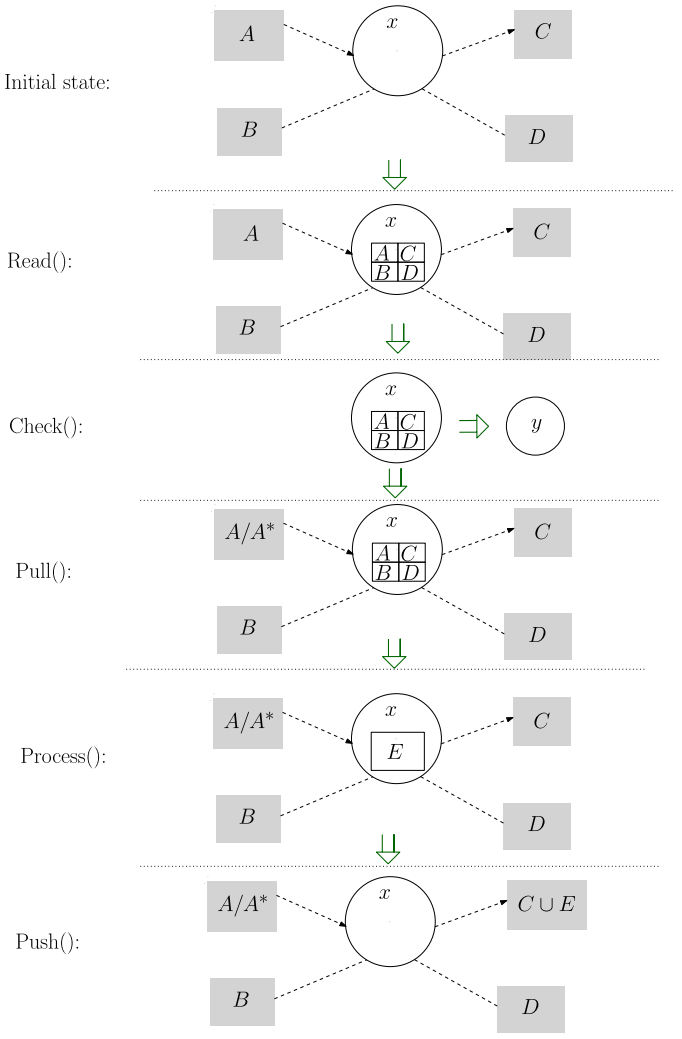

Figure 9. Summary of movement and processing of information done by transition functions in a node.

compute summary statistics and changes to the environmental variables, and it can *push*() to commit those changes back to the environment.

Termination: Terminate execution. It does nothing, so it does not use any of the component transition functions.

5.3 Container Nodes

Container nodes are interfaces between control nodes and the things in the system, rather than control elements themselves. We prevent any modification of objects inside container nodes, to preserve this separation. Every container node has three functions forming its uniform interface: *read*(), *add*(), and *remove*(). These are used respectively by the *read*(), *push*(), and *pull*() functions of control nodes. Internal data can be organized in any way the node designer sees fit as long as it provides these three functions. An implementation could move from using a list to a database by changing only the container, and not need to make any change to control nodes using it.

Table 2. Transition functions used by different types of control nodes; unchecked functions always use their default behavior.

	Action	Decision	Sample	Observer	Termination
read	✓	✓	✓	✓	
check	✓				
pull	✓		✓	✓	
process	✓	✓		✓	
push	✓		✓	✓	

Container nodes are partitioned into two subtypes: particle container nodes and environment container nodes.

Particle container node: contains a multiset (bag) of particles; the contents of this multiset change as the AChem executes. The state of all the containers in the system partitions of the particles in the system. There are two subtypes of particle container nodes: samples and tanks. Tanks are protected containers. Particles in tanks can be moved in and out but cannot be changed in the tank; any changes must be made in sample containers, so the designer has to decide what will be changing.

Examples. A beaker being used for an experiment, a pipette, a petri dish.

Environment container node: contains non-particle objects and information in the system. The system can have multiple environments, to make reference to the things in the environment easier. For example, one might want to store a time record separately from summary statistics or log information. These are all still accessed via a mapping from the node and state of the system to the dynamic information and objects.

Examples. Temperature readings, Bunsen burner, stirrer, observation log.

The limits on access to containers placed on control nodes are given in Table 3. Any control node can *read*() any container node (information is always knowable). However, we limit the modification of containers to certain types of control nodes, to make it easier to track activity in the system. This should also encourage limiting the scope of individual nodes to a basic action that may be reusable.

5.4 Examples from StringCatChem

5.4.1 Local State

Here we give examples of the behavior of some of the control nodes in StringCatChem. These involve reading particles and environment variables into a local state. This local state is modeled

Table 3. The container nodes that can be modified by a control node (by *push* or *pull*); *read* can be performed on any container.

	Action	Decision	Sample	Observer	Termination
Tank			✓		
Sample	✓		✓		
Environment	✓			✓	

(see Appendix, equation 31) as a pair of mappings, the first from particle (tank and sampler) node names to contents, the second from environment node names to contents.

5.4.2 Sampler Node

Here we describe the micro-level **s**:sampler from Figure 8. Sampler nodes have *read*(), *pull*(), and *push*() functions (Table 2). In this example, this sampler randomly selects a single particle from a tank to move to a sample container.

***read*()**: Read the contents of the containers attached by information edges. The node **s**:sampler has two read edges, one to **T:**tank (also a pull edge) and one to **S:**composite (also a push edge). After the read, the local state Λ has a copy of the states of these two particle containers (defined in the global state G); there are no connected environment containers, so it has an empty environment component:

$$G = (\{\mathbf{S}\text{:composite} \mapsto \Sigma, \mathbf{T}\text{:tank} \mapsto T, \ldots\}, \{\ldots\})$$

$$\Lambda = (\{\mathbf{S}\text{:composite} \mapsto \Sigma, \mathbf{T}\text{:tank} \mapsto T\}, \{\})$$

where Σ is (a copy of) the particles in **S:**composite, and T is (a copy of) the particles in **T:**tank.

***pull*():** Select a random particle τ from T, and delete (pull) the corresponding particle from the external container **T:**tank:[3]

$$G = (\{\mathbf{S}\text{:composite} \mapsto \Sigma, \mathbf{T}\text{:tank} \mapsto T \setminus \{\tau\}, \ldots\}, \{\ldots\})$$

$$\Lambda = (\{\mathbf{S}\text{:composite} \mapsto \Sigma, \mathbf{T}\text{:tank} \mapsto T\}, \{\})$$

***push*():** Push the selected particle τ to the **S:**composite sample:

$$G = (\{\mathbf{S}\text{:composite} \mapsto \Sigma \cup \{\tau\}, \mathbf{T}\text{:tank} \mapsto T \setminus \{\tau\}, \ldots\}, \{\ldots\})$$

$$\Lambda = (\{\mathbf{S}\text{:composite} \mapsto \Sigma, \mathbf{T}\text{:tank} \mapsto T\}, \{\})$$

On moving to the next control node, the local state Λ is destroyed, with the overall result that particle τ has moved from the sampler to the tank.

5.4.3 Observer Node

Here we describe the macro level **o**:time from Figure 6. Observer nodes have **read**(), **pull**(), **process**(), and **push**() functions (Table 2). In this example, this basic observer increments a variable representing time.

read(): Read the contents of the containers attached by information edges. The node **o**:time has one read edge (also a pull and a push edge), to the environment container

3 For notational simplicity in these examples, we assume here that containers contain *sets* of particles; in the full formalism, the containers contain *multisets* (bags) of particles (Appendix A.2), allowing multiple instances of a given species.

V:time. After the read, the local state Λ has a copy of the state of this environment container; there are no connected particle containers, so it has an empty particle component:

$$G = (\{\ldots\}, \{\mathbf{V}\text{:time} \mapsto \mathcal{V}, \ldots\})$$

$$\Lambda = (\{\}, \{\mathbf{V}\text{:time} \mapsto \mathcal{V}\})$$

where $\mathcal{V}$ is (a copy of) the environment in **V**:time. Here, the environment contains a single variable, representing the time.

pull(): As our **V:**time container only contains a single variable, our pull function clears the **V:**time container:

$$G = (\{\ldots\}, \{\mathbf{V}\text{:time} \mapsto \emptyset, \ldots\})$$

$$\Lambda = (\{\}, \{\mathbf{V}\text{:time} \mapsto \mathcal{V}\})$$

We remove the value from the attached container, because interactions with the container are limited to **read**(), **add**(), and **remove**() functions; if we wish to update or modify a variable, we must read it in, remove it from the container, and then add the new version. If we simply push (add) the new version without clearing the old one, behavior is undefined and will depend on implementation.

process(): This observer is a *counter observer*: It increments a single variable. In this case the variable is time and the increment is 1. This is performed on the variable $\mathcal{V}$ in local storage:

$$G = (\{\ldots\}, \{\mathbf{V}\text{:time} \mapsto \emptyset, \ldots\})$$

$$\Lambda = (\{\}, \{\mathbf{V}\text{:time} \mapsto \mathcal{V} + 1\})$$

push(): Push the incremented local variable back out into the **V:**time container for storage:

$$G = (\{\ldots\}, \{\mathbf{V}\text{:time} \mapsto \mathcal{V} + 1, \ldots\})$$

$$\Lambda = (\{\}, \{\mathbf{V}\text{:time} \mapsto \mathcal{V} + 1\})$$

On moving to the next control node, the local state Λ is destroyed, with the overall result that the environment variable **V**:time has been incremented by one.

5.4.4 Action Node

Here we describe the micro level **a**:split from Figure 8. Action nodes can use all five component transition functions. In the case of **a:**split we explicitly use some of these functions, and use the default definition of the others.

read(): The **a:**split action node has an information edge to only one other node, in the form of a read, pull, and push edge between it and **S:**composite. At this stage

S:composite holds a single particle string, which contains a double letter. This string is read into the local particle container:

$$G = (\{\textbf{S}\text{:composite} \mapsto \{\text{“prexxpost”}\}, \ldots\}, \{\ldots\})$$

$$\Lambda = (\{\textbf{S}\text{:composite} \mapsto \{\text{“prexxpost”}\}\}, \{\})$$

check(): In this particular system reactions are deterministic, so we use the default behavior of the check function, which is to return zero. So the check test is true, and the rest of the action happens.

pull(): Delete (pull) the string from the **S**:composite node:

$$G = (\{\textbf{S}\text{:composite} \mapsto \emptyset, \ldots\}, \{\ldots\})$$

$$\Lambda = (\{\textbf{S}\text{:composite} \mapsto \{\text{“prexxpost”}\}\}, \{\})$$

process(): Process the contents of the local state. Here, divide the string at the double letter, and store the two resulting strings in the local particle state:

$$G = (\{\textbf{S}\text{:composite} \mapsto \emptyset, \ldots\}, \{\ldots\})$$

$$\Lambda = (\{\textbf{S}\text{:composite} \mapsto \{\text{“prex”}, \text{“xpost”}\}\}, \{\})$$

push(): Push the resulting strings back into the **S**:composite sample node:

$$G = (\{\textbf{S}\text{:composite} \mapsto \{\text{“prex”}, \text{“xpost”}\}, \ldots\}, \{\ldots\})$$

$$\Lambda = (\{\textbf{S}\text{:composite} \mapsto \{\text{“prex”}, \text{“xpost”}\}\}, \{\})$$

On moving to the next control node, the local state Λ is destroyed, with the overall result that the **S**:composite sample node now contains the split strings.

5.4.5 Decision Node

Here we describe the micro-level **d:**decomp from Figure 8. Decision nodes use two of the five transition functions. They do not change the state of any of the containers (so there is no pull or push). They just **read** containers, and compute (*process*) the next control node to move to.

read(): The **d:**decomp decision node has a read edge to between it and **S:**composite. At this stage **S:**composite holds a single particle string, which may or may not contain a double letter. This string is read into the local particle container:

$$G = (\{\textbf{S}\text{:composite} \mapsto \{\text{“}p_1p_2..p_n\text{”}\}, \ldots\}, \{\ldots\})$$

$$\Lambda = (\{\textbf{S}\text{:composite} \mapsto \{\text{“}p_1p_2..p_n\text{”}\}\}, \{\})$$

process(): This function performs the computation that makes the decision. It returns one of the possible next control nodes, **s:**sampler or **a:**split, and stores this in the local state for the *next* function to access:

$$G = (\{\mathbf{S}\text{:composite} \mapsto \{\text{“}p_1 p_2 .. p_n\text{”}\}, \ldots\}, \{\ldots\})$$

$$\Lambda = (\{\mathbf{S}\text{:composite} \mapsto \{\text{“}p_1 p_2 .. p_n\text{”}\}\}, \{\mathbf{V}\text{: local} \mapsto c\})$$

where

$$c = \begin{cases} \mathbf{a}\text{:split} & \text{if } \exists i \in 1..n-1 \mid p_i = p_{i+1} \\ \mathbf{s}\text{:sampler} & \text{otherwise} \end{cases}$$

On moving to the next control node, c, the local state Λ is destroyed, with the overall result that no containers have changed state, and the control node is the relevant one for the string in container **S:**composite.

6 Jordan Algebra Artificial Chemistry

MetaChem can be used to design new AChems, and to describe existing AChems. Here we use MetaChem to describe our earlier Jordan algebra artificial chemistry (JA-AChem) [6]. We choose this as one extreme of an artificial chemistry. It is subsymbolic in that its bonding properties arise from the internal structure of its particles, and is designed to work at the level of atoms. It has both a linking action and a destructive action, which makes its algorithm more complicated than that of the StringCatChem already described.

6.1 Overview of Particles and Linking

Hermitian matrices provide a rich variety of properties such that we can use them as prime material for creating a subsymbolic AChem [5], where emergent properties of the matrices dictate the linking capabilities and probabilities of a particle, and the algebra gives the structure of the composite particles. Here we give a condensed overview of the properties used to form atoms and composite particles; in the next section we use these definitions in the overall MetaChem description of the JA-AChem.

6.1.1 Definitions and Properties

The complex conjugate of the transpose of a matrix M is written as $M^\dagger$. A matrix M is Hermitian if it is equal to the complex conjugate of its transpose: $M = M^\dagger$.

The Jordan product of two square matrices is

$$A \circ B := \tfrac{1}{2}(AB + BA) \tag{1}$$

Hermitian matrices are closed under the Jordan product [15].

The eigenvalues λ and eigenvectors $\mathbf{v}$ of a square matrix M are solutions to $(M - \lambda I)\mathbf{v} = 0$. An nD matrix has n (possibly degenerate) eigenvalues, and n corresponding eigenvectors. The eigenvalues of a Hermitian matrix are all real.

6.1.2 Atoms

The atoms in the Jordan algebra AChem used here are specific 3×3 Hermitian matrices. The atom set is

$$\left\{ A = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{pmatrix} : x_{ij} \in \{\pm 1, \pm i, \pm 1 \pm i, 0\} \right\} \tag{2}$$

We use the eigenvalues of the matrices to define linking properties.

There are 14574 atoms, with 66 different sets of eigenvalues. We have many options, and many different sorts of operations and linking behaviors are possible.

6.1.3 Composite Particles

We use unit eigenvectors $\hat{\mathbf{v}}_i$ and their corresponding normalized eigenvalues μ_i to define linking probabilities:

$$\mu_i = \lambda_i / \sum \lambda_j \tag{3}$$

We normalize the eigenvalues to ensure sensible linking probabilities of larger composites. (The sum of the eigenvalues equals the trace of the matrix, which is required to be nonzero in this system.)

We define the *alignment* of two eigenvectors (one from each particle's matrix) as

$$a_{ij} = \left(1 - \tfrac{1}{2}\left(\left(\hat{\mathbf{v}}_i \cdot \hat{\mathbf{v}}_j\right) + 1\right)\right) \tag{4}$$

This definition uses the dot product between unit vectors, which is the cosine of the angle between them. Hence the alignment has a value between 0 and 1, being 0 if the vectors are parallel and 1 if they are antiparallel.

When linking two particles A and B, we calculate the normalized eigenvalues μ_{A_i}, μ_{B_j} and the corresponding unit eigenvectors $\hat{\mathbf{v}}_{A_i}$, $\hat{\mathbf{v}}_{B_j}$. We calculate all the alignments between pairs of eigenvectors, one from each particle. We choose the highest alignment value (the most antiparallel eigenvectors) as the linking eigenvectors (i, j).

We calculate the *strength* of this alignment, using the corresponding eigenvalues:

$$s_{A_i B_j} = \mathcal{N}\left(\mu_{A_i} - \mu_{B_j}\right) \tag{5}$$

where $\mathcal{N}(x)$ is the value of the probability density of the normal distribution ($\mu = 0$, $\sigma = 1$) at x. This will give a probability of linking that is larger for more similar normalized eigenvalues. The normal distribution is not the only option we could use here; we investigate other options in [7].

We calculate the probability of linking based on the strength of the link and its alignment.

$$p_{AB} = s_{A_i B_j} a_{A_i B_j} \tag{6}$$

If the link is formed, the resulting composite particle is the Jordan product of the components.

6.2 Macro Description of Jordan Algebra Artificial Chemistry

In addition to these particles, we need an algorithm for how our system works. This covers not only the linking and decomposition aspects, but the entire behavior of our chemistry. We define this using MetaChem, starting at the macro level, which describes the overall behavior of the system

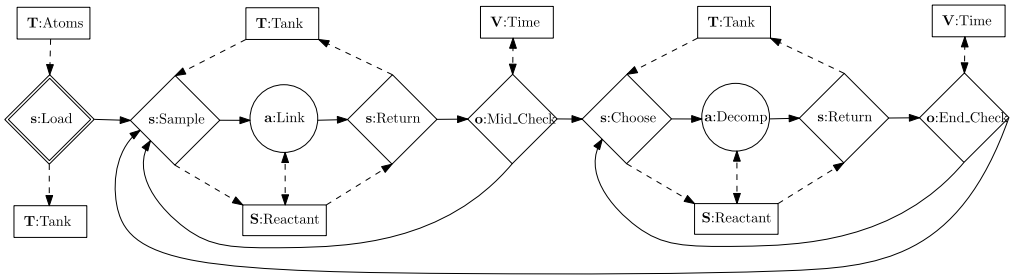

Figure 10. Macro-level description of JA-AChem operating over link and decomp loops.

over time. We then look in more detail at the linking process. For more information on how decomposition works see [6].

The algorithm loads the initial atom or particle set and then operates over two loops (Figure 10). These two loops are similar, starting with sampling from the tank followed by an operation. The first loop performs linking; the second loop performs decomposition. The loops finish by returning their modified samples to the tank, updating timing variables, and checking if enough operations or time has passed to say whether the loop continues or the system moves to the other loop.

If we make observations of our system, we add them to our logger, which can be added to the system in Figure 10. The logger pushes to an external environment, which is never pulled from. These observations can provide many different summary statistics. In later examples relating to linking and probabilities we observe and log the number of atoms in each particle, the number of different atoms in each particle, the size of particle traces, the weight of particles, and the size of largest link in particles.

This macro-system-level description does not cover the internal workings of our link and decomposition nodes. The algorithms for these are described in detail in [6].

6.3 Micro-Level Description of Linking in JA-AChem

Now we have the wider view of how this AChem works, we look in more detail at the micro-level description of **a:**Link. This is defined by the MetaChem graph in Figure 11. It has been defined and labeled in terms of the macro-level component transition functions in **a:**Link. We discuss the process in terms of these components below.

The **a:**Link action node uses all five component transition functions. The read, pull, and push functions are all performed from and to the **S:**reactant sample. More interesting in this case are the check and processing functions. To describe these actions in more detail, we expand the **a:**Link action node into a micro-level subgraph of the macro system. This means that all five transition functions are themselves defined in terms of micro-level graphs, and we introduce various explicit micro-level containers **V**:xxx, **T**:xxx, **S**:xxx to implement the macro-level local state.

Some of these subgraphs do processing where the high-level component function does not. For example, processing does not occur in the macro-level **read**() function, but does in the corresponding part of the micro-level graph. Now that we are taking a lower-level view of this function, we can reveal more of the implementation, and with it the shortcuts we take to reduce repetition of calculations and have a smoother flow. So this lower-level description is not a formal refinement of the individual component functions, but involves some refactoring (rearranging) of the functionality.

6.3.1 Expansion of Macro-Level read()

Three observers, **o:**internal_struct, **o:**Alignment, and **o:**Strength, gather the information needed for the linking probability check. They do this by reading information from **S:**reactants and processing it. The information about the internal structure and other derived values is stored in various environment containers **V**:Mat, **V**:Eval, **V**:Evec, **V**:Pairs, **V**:Strengths.

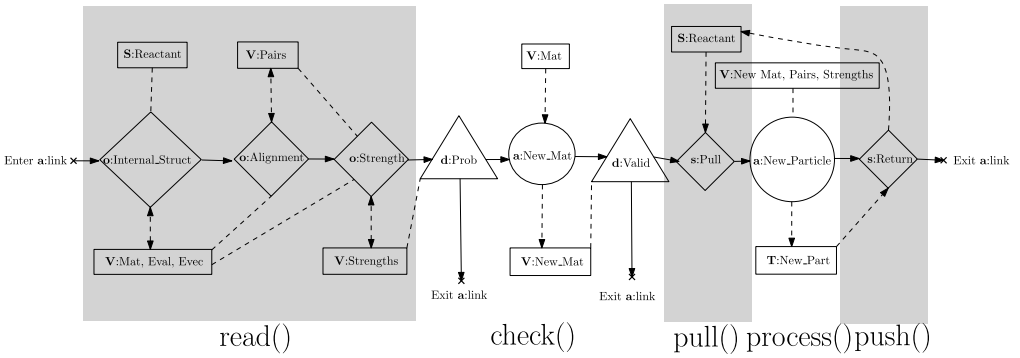

Figure 11. Micro-level description of **a:**link node from Figure 10. Below are the labels for the grouping of control nodes according to transition function in the outer node. The only external container used here is **S:**Reactant. All other containers are part of L_V and are lost at completion of control flow. We have no need here for L_B.

The observer **o:**internal_struct pulls any previous matrices, vectors, and eigenvalues from the container to clear it. It then extracts the matrix, normalized eigenvalues, and unit eigenvectors for each particle in the sample reactants and pushes these to the environment container.

The observer **o:**Alignment reads those values, then clears any preexisting pairs, and then uses the new values to calculate the best pairs of eigenvalues to use, based on their alignment (Equation 4).

The observer **o:**Strength clears the previous strength value and then uses this new information to calculate the strength (Equation 5) between the (i, j) pair selected by the alignment, and stores it in the container **V:**Strengths for use in the next phase.

6.3.2 Expansion of Macro-Level check()

The macro-level **check**() function in the **a:**link node looks like

$$\begin{cases} pass & \text{if } r < p_{AB} \ \& \text{ resultant matrix is valid} \\ fail & \text{otherwise} \end{cases}$$

where p_{AB} is the linking probability (Equation 6), and r is a uniformly distributed random value on the interval [0:1].

In the micro-level view we break this down into two decisions. The first, **d:**Prob, decides whether to progress based on the probability of linking,[4] $r < p_{AB}$.

If the probability check passes, we perform some micro-level processing. We use a deterministic action (one that always runs in full) **a:**New_Mat to calculate what the new matrix would be (Equation 1) if we did create the new particle. We store this for possible later use in **V:**New_Mat.

The second decision, **d:**Valid, uses this result to make its decision based on the trace of the new matrix. If the matrix has a nonzero trace, we continue to **s**:Pull; otherwise, we exit the macro-level **a:**link node without ever pulling information, and without destroying existing particles or creating new ones, leaving **S**:reactant unchanged.

6.3.3 Expansion of Macro-Level pull()

This single-node action, **s:**pull, just empties the existing reactants out of the **S:**reactants container. We delete them by pulling them from the container, so they are no longer accessible.

4 Here we have only a single probability, for linking two particles. Full JA-AChem has a more sophisticated approach, allowing multiple particles to link via higher-order Jordan products, which involve multiple probabilities.

6.3.4 Expansion of Macro-Level process()

This single node, **a:**New_Particle, uses the previously calculated matrix stored in **V**:New_Mat to create a new particle. To do this it creates a link object with the previously calculated strength from **V:**Strengths. This link object includes a memory of the reactants that are used to form the link, taken from the local environment container. These properties, the list of eigenstate pairs used, and the matrices are used to generate a new particle object. The new particle is stored in its own labeled section of the the local storage, **T:**New_Part.

6.3.5 Expansion of Macro-Level push()

The sampler **s:**return moves the newly formed particle in **T:**New_Part into the empty **S:**Reactants. Control then exits the node.

6.3.6 Clearing Local Containers

If this functionality were implemented as the macro-level node, then the local containers would be deleted when control leaves the node. In this micro-level implementation we have global containers that persist. We could add an additional observer node just to clear these containers before leaving this section. In the current implementation we instead give the observers pull access to these containers, and they pull (clear) the variables before they push new values.

6.4 Modifying JA-AChem

JA-AChem was first introduced in [6]. That JA-AChem system has two properties of its algorithm that we modify here.

6.4.1 Mass Conservation

The original JA-AChem has no analogue of mass conservation: Particles that react are not removed from the system. So it explores the space of possible composite particles with no limitation of resource, always with a plentiful supply of all discovered particles. Secondly, the reactions take place in a single well-mixed tank, with no spatial component.

In the classical (S, R, A) model of AChems, any change to the algorithm to change these features is considered to create a different chemistry (to some degree). In MetaChem, we can change some features at a macro level without changing the micro-level detailed linking and reactions: We can change the "glassware" without changing the "chemicals."

We have mass conservation in the JA-AChem described above. When a link is formed, the components used to make the new particle are pulled from the tank (rather than merely being read) and replaced with the new particle. Similarly, when a link decomposes, the components of the link and any remnants of larger particles are processed and returned to the tank. In this way the total numbers of atoms both free and bound within particles remain constant.

6.4.2 Multiple Tanks

We use MetaChem to introduce multiple tanks, and allow them to interact by transferring particles between them. We do this by adding a new outer loop to the overall macro-level flow, similarly to the approach described in StringCatChem (Figure 7).

When transferring particles between two tanks, we take the contents of both tanks, sort them by size (number of atoms in a particle), and then return them to the two tanks by starting with the largest and returning it to a tank and returning the subsequent particles to whichever tank contains less atoms in total. This maintains a rough equality in the number of atoms in each tank.

In Section 8 we consider a single tank, multiple noninteracting tanks, and tanks that interact in a grid or in a random organization.

There are no transfers when working with a single tank (though we do use a larger single tank with the same number and composition of atoms in the combined contents of the multiple tanks). In multiple-tank systems where we choose to have interactions, we do this in two ways. In both

cases we choose a random number of transfers within a range (in this case 0 to 10 transfers). In the first case (random transfers) we sample two tanks without replacement from all the tanks in the system and perform a transfer. In the second (grid transfers) we choose a single tank at random and then select the second tank from the Moore neighborhood around the first tank.

7 Swarm Chemistry

We have developed a framework for describing artificial chemistries to replace the limited (S, R, A) format. However, all the chemistries we have described in the new framework so far could be built within (S, R, A).

In this section we describe Swarm Chemistry [21], a system built to explore beyond the (S, R, A) format. Despite not having a comfortable description in the (S, R, A) framework—it does not have direct interactions between particles—SwarmChem is widely known and accepted as an artificial chemistry. It is therefore important to show that, while (S, R, A) may struggle with SwarmChem, MetaChem comfortably describes it. In the description of SwarmChem in MetaChem we present here, we can see that SwarmChem is not some borderline AChem. It is seen to have many close similarities to other, more classical AChems when we consider its controls and algorithms, rather than simply its lack of physical connections.

7.1 Flocking in SwarmChem

The individuals in SwarmChem, often referred to as *boids* or agents, interact by each boid changing its own velocity based on the local positions and velocities of its neighbors. This involves no knowledge of the neighbors' internal parameters, just observation of their velocities and positions. This gives the effect of swarming or flocking like that seen in birds. Different parameters sets produce swarms that differ in their density and in how they move. In SwarmChem boids with different parameters are allowed to mix (Figure 12).

SwarmChem is a framework for a class of artificial chemistries. Its intention is to explore how higher-level statistical rules for chemical systems emerge from lower-level local interactions. It does this with the basic concepts of [20]'s boids.

Flocking in both boids and SwarmChem works as follows: At each time step for each boid we first work out the neighborhood of the boid. We then calculate an acceleration vector of the boid towards the center of the group of neighboring boids; this is called the *cohesion*. We then calculate a

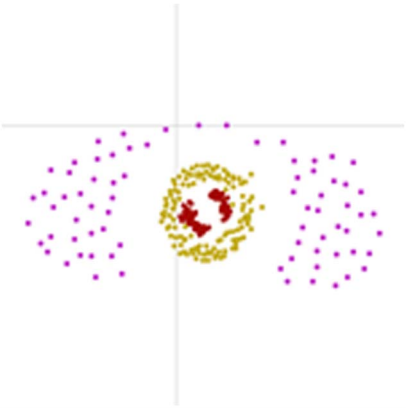

Figure 12. Pulsating eye swarms contributed to SwarmChem by Benjamin Bush using the recipe: 102 * (293.86, 17.06, 38.3, 0.81, 0.05, 0.83, 0.2, 0.9) 124 * (226.18, 19.27, 24.57, 0.95, 0.84, 13.09, 0.07, 0.8) 74 * (49.98, 8.44, 4.39, 0.92, 0.14, 96.92, 0.13, 0.51), from bingweb.binghamton.edu/sayama/SwarmChemistry/. An example of interesting 2D organization using three different parameter sets for 300 boids.

vector toward the average heading of the neighboring boids; this is called the *alignment*. We then calculate a vector to prevent crowding, moving to increase the *separation* between boids. Finally we perform *pacekeeping*, which biases the pace (speed) of the boid towards its normal speed in order to prevent all boids' either becoming stationary or tending towards their maximum speeds. Then the boid is moved, based on this information. This is done on all boids at once, so we use the information of position and velocity from the current time step to calculate the next. See Figure 13 for a visual description.

The key change SwarmChem makes to the boid system is to assign *recipes* (parameter sets) to individual boids, rather than using global fixed values. This allows heterogeneous swarms, which can form new kinds of the patterns through their interactions.

In the basic SwarmChem framework, each boid operates based on a particular recipe; extended versions may use multiple recipes with weights used to choose the active recipe [22–24]. Recipes and weights can be exchanged and changed by other boids. This process can be based on collision or other factors.

This exchange gives boids a mechanism to change and optimize to maintain structures. This allows a form of evolution, if we consider a boid to be a child of itself when its parameters change.

More recently this system has been extended by identifying these larger structures and considering them as entities in their own right [26]. Such an approach is important in the analysis of multilevel artificial chemistries. Initially, the boids were restricted to 2D space, but another extension places the boids in a 3D space [25].

Here we describe a variant of SwarmChem in MetaChem. In this variant we exchange a random number of parameters when a collision occurs. This is different from the weighted recipe method used elsewhere. In other versions one boid is dominant in the collision and enforces its recipe on the other, whereas collisions in our system are "no fault," in that both boids are changed. We also prevent trading normal or max parameters if that would result in the boid's normal exceeding its max. A preliminary version of this description can be found in [18]. Here we describe it using a full macro-level graph, together with a micro-level graph of the update process, an expansion of the flock action. We use this description in the nested AChem described in the next section. This demonstrates that SwarmChem fits comfortably in the MetaChem framework, and can be combined with other AChems.

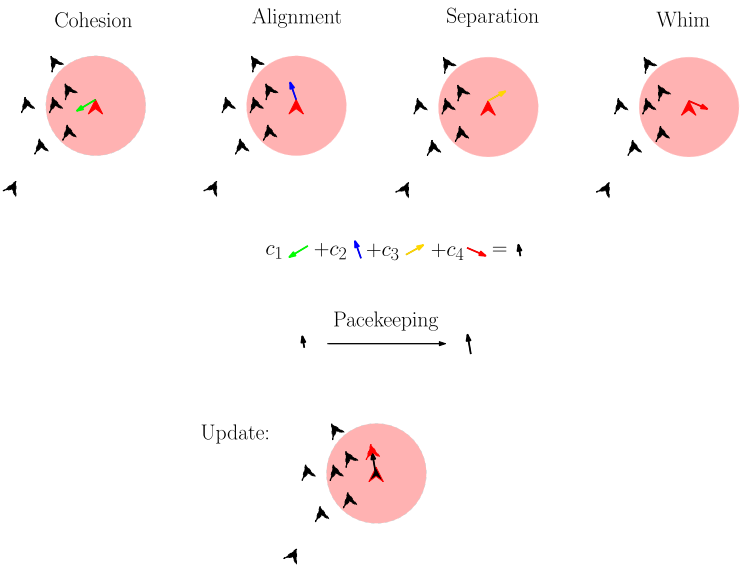

Figure 13. A pictorial description of flocking in Reynolds' boids and Swarm Chemistry. The red disk shows *R*, the perception distance of our boid.

7.2 Macro Description

The macro-level graph of our variant of SwarmChem is shown in Figure 14. It operates as follows:

- **s**:Load Parameters: Starting node, which loads the initial parameter set from T:Parameters, and randomly positions the boids, stored in **S**:*n*.
- **o**:Generation: Iterate the clock. This "tick" is part of the discrete timing system that is consistent with all current swarm systems. This is evident in the rest of the macro system as well.
- **s**:Copy_to_Previous: The sampler copies the current generation from **S**:*n* to the tank **T**:*n*−1, which is used to hold the previous generation. This gives a copy of the previous state of all the boids for use in subsequent calculations.
- **a**:Flock: Update each boid's parameters (stored in **S**:*n*) by following the classic boid rules.
- **a**:Move: Move all the boids (stored in **S**:*n*) based on their parameters and current headings and velocities. This is common to all swarm chemistries.
- **o**:Collisions: Check for collisions, and record them in **V**:Collisions. This is part of our variant SwarmChem.
- **a**:Update_Params: Update parameter sets that are changed by collision. **S**:*n* now contains the fully updated generation.
- **s**:Log: Log the previous generation to **T**:external; clear **T**:*n*−1 ready for the current generation to be copied in the next loop iteration.
- Loop back for the next iteration.

7.3 Micro Description of Flocking

The flocking action captured in the macro-level **a**:Flock node (Figure 14) contains most of the activity of the system. In this section we expand that node in a micro-level graph, Figure 15.

As with the JA-AChem example, we do not formally refine each macro-level component function individually, but rather use that structure to guide the design of the micro-level description. Here we sequentialize the operation, by performing the composition of component functions on each individual particle in the swarm.

7.3.1 read()

We start by reading various individuals into different tanks. The sampler **s**:Update_Boid reads a random single boid from **S**:*n* into **S**:boid for updating. The sample **s**:Find_Neighbours reads out all the neighbours of this boid, defined by its perception distance, into **S**:Neighbours. The observer o:Local_Averages generates $(\bar{v})$, $(\bar{x})$, and $(\bar{s})$ of the boids in the neighborhood, and stores them in the environment **V**:Averages.

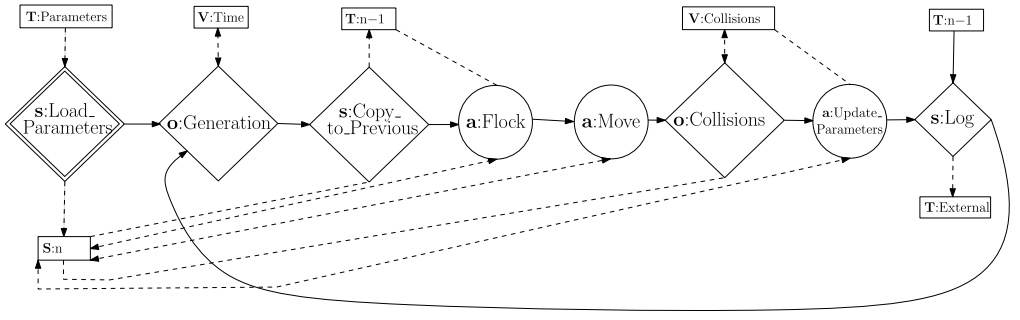

Figure 14. Macro-level Swarm Chemistry graph. It includes the timing counter **o:**generation, flocking, and moving as well as collisions and the logging sampler to track the chemistry.

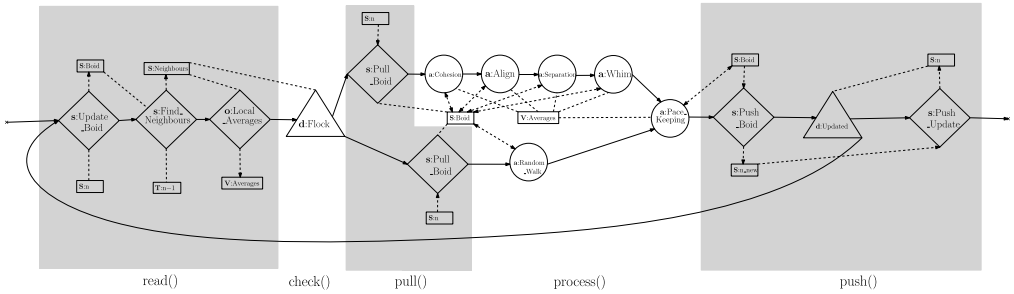

Figure 15. Micro-level description of flocking in Swarm Chemistry. Due to the large loop in this, we could consider it a description of many identical nodes or of one node at the macro level.

7.3.2 check()

The action always occurs, so the macro-level check always returns true. Here we choose to implement a decision to decide the actual process of the function, which is an analogous choice. The decision node **d**:Flock makes a choice of performing a random walk or normal flocking behavior, based on the number of neighbors. If $|N| > 0$ then the operation is flocking, else a random walk.

7.3.3 pull()

The sampler s:Pull_Boid removes the selected boid from the current generation **S**:*n*. In a subsequent refactoring, we might merge this into the B:Update_Boid node to simplify the graph. However, here our initial design is being guided by the transition function format.

7.3.4 process()

There are two processing paths.

In the random walk path, a:Random_Walk sets the current boid with a random velocity given by

$$\text{Straying:} \quad a_i = \left(r_{\pm s}, r_{\pm s}\right) \tag{7}$$

Along the flocking path, four action nodes perform the four calculations of cohesion, alignment, separation, and whim (a small random component added to the motion to keep the system from behaving too predictably) as follows: **a**:Cohesion implements Equation 8, **a**:Alignment implements Equation 9, **a**:Separation implements Equation 10, and **a**:Whim implements Equation 11:

$$\text{Cohesion:} \quad a_i = c_1\left(\bar{x} - x_i\right) \tag{8}$$

$$\text{Alignment:} \quad a_i = a_i + c_2\left(\bar{v} - v_i\right) \tag{9}$$

$$\text{Separation:} \quad a_i = a_i + c_3\bar{s} \tag{10}$$

$$\text{Whim:} \quad a_i = a_i + \left(r_{\pm s}, r_{\pm s}\right) \tag{11}$$

The branches rejoin at this point, and **a**:Pacekeeping implements the remaining equations:

$$\text{Acceleration:} \quad v_i^* = v_i + a_i \tag{12}$$

$$\text{Prohibit overspeeding:} \quad v_i^* = \min\left(v_m / \left|v_i^*\right|, 1\right) \cdot v_i^* \tag{13}$$

$$\text{Pacekeeping:} \quad v_i^* = c_5\left(v_n / \left|v_i^*\right| \cdot v_i^*\right) + (1 - c_5)v_i^* \tag{14}$$

The intent here is to prevent boids from constantly increasing in speed, by modifying their speed back towards their normal velocity v_n.

7.3.5 push()

Having finished processing, **s**:Push_Boid pushes the processed boid to a different sample, **S**:*n*_new, which keeps track of the boids that have been processed. The decision **d**:Updated decides whether to loop back to process a further boid, or to continue, depending on whether the generation sample **S**:*n* has been emptied yet. Once all boids have been processed, **s**:Push_Update moves them from **S**:*n*_new back into the (now empty) **S**:*n*.

Note that within this graph we do not update the velocity; this is done along with position in the macro-level **a**:Move node.

7.4 Example: Pulsating Eye

With this implemented in MetaChem we can still work with the same recipes already generated for Swarm Chemistry. For example here we implemented the recipe shown in Figure 12. In this case (Figure 16), we get spinning sets of pulsating eyes, which merge into larger and larger eyes. This behaves a little differently because we have a different set of collision rules. But it is visibly the same recipe made of three different parameter sets.

8 Nested Chemistries

8.1 Levels of Chemistries

Subsymbolic artificial chemistries (ssAChems) [3–5] are generally AChems whose atoms and particles have internal structure that determines their behavior. JA-AChem is one such ssAChem:

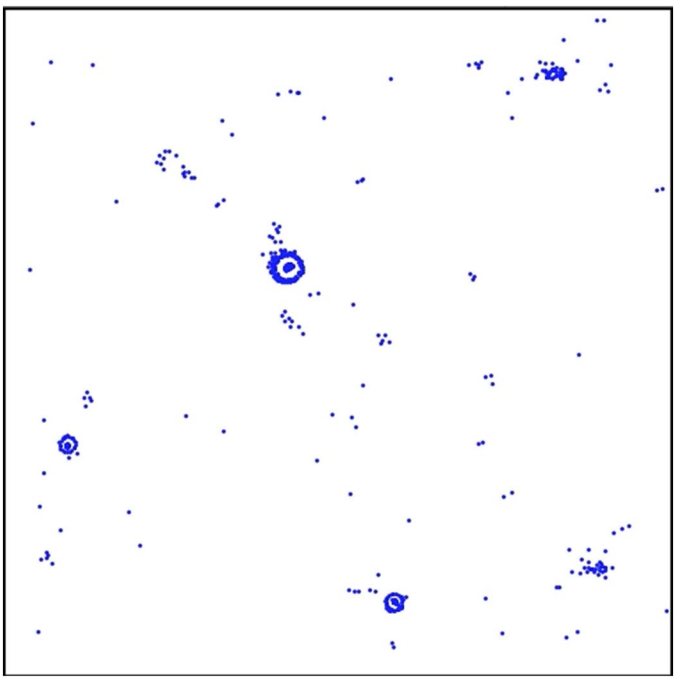

Figure 16. Pulsating eye recipe as implemented in MetaChem.

Particles are matrices whose internal structure (elements) determines their linking behavior through eigenvalues and eigenvectors. The existing ssAChems are analogous in their rationales to natural chemistry viewed at the level of atomic structure affecting molecular properties.

Other AChems are designed to reflect the properties of chemistry at the level of cells [11, 14] or chemical reaction systems [29]. In natural chemistry these different levels are closely related: Cells contain chemical reaction systems, and chemical reaction systems are based on individual particle and atom interactions. While attempts have been made to bridge the gaps between such levels in individual systems [13], so far the systems have been very simple and lacking in more complex features.

We can take advantage of feature-rich existing AChems by using MetaChem to combine them to give a system that can span different levels of activity and behavior in a single AChem system. We demonstrate this approach here by combining JA-AChem [6, 7] and SwarmChem [21, 23, 24, 28].

8.2 General Method

We can connect any two AChems in MetaChem by giving them the ability to communicate via their environment [18]. This communication can be uni- or bidirectional. The basic MetaChem graph structure of combined communicating AChems is given in Figure 17.

We use color to indicate the *ownership* of a node by a single system. We do not allow a node owned by one AChem to directly communicate with the nodes owned by a different AChem. Instead, information is shared using an environmental container that is not owned by either AChem. So in Figure 17, the blue observation is of a tank in the blue AChem, and the pink action is on a tank in the pink AChem. The figure shows unidirectional communication, in which the blue AChem influences the pink. By adding a second link in the other direction we could establish bidirectional communication. Both the action and the observation are defined by the designer.

For example, if we wish to establish *side-by-side* chemistries, where two chemistries with their own separate particles and reactions coexist in the same spatial system, then our observation will produce a summary statistic that is a value, or set of values, based on the entire system in the first chemistry, which will uniformly affect the entire system in the second chemistry. Alternatively, the observer can generate statistics based on individual particles, which can then affect individual particles in the second AChem.

Below we give an example of a *nested*, or multilevel, AChem with bidirectional communication. The observer of the lower-level AChem generates a set of values over a large number of particles in that AChem. These values are then used to influence the behavior of a single particle in the higher-level AChem. In turn, the behavior and interaction of one or two particles in the higher-level AChem influence a large number of particles in the lower-level AChem.

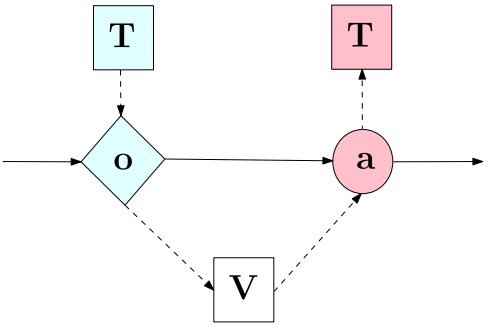

Figure 17. Communication link between the blue AChem and the pink AChem. The observer node observes the blue AChem's tank and pushes the communicated information to the shared environment. Control passes to the pink AChem's action node, which acts on its tank based on information read from the shared environment.

8.3 Implementation: Nested Chemistry

We generate a new set of chemistries by combining JA-AChem and SwarmChem; each of the particles of SwarmChem contains a well-mixed tank of JA-AChem particles (so we have swarming tanks of matrix particles), whose properties inform the SwarmChem particle parameter values. SwarmChem's spatial movement provides a limitation and control on particle exchanges in JA-AChem between different tanks. The JA-AChem tanks communicate with SwarmChem by changing its parameter values, which influences the agents' spatial movement and likelihood of collision.

First, we abstract the description of both AChems to a higher level that comprises two control nodes and two container nodes. The first control node, **s**:LoadX, is the initialization sampler, which loads the initial state from **T**:InitX into **T**:X. The other control node, **a**:UpdateX, performs one of the outer loops of the AChem's operation as defined in the earlier macro-level graphs.

The other control nodes associated with each of our separate chemistries is new and deals with modifying the associated chemistry according to information observed from the other chemistries' particles.

We link these individual high-level AChem graphs in various ways to give seven distinctive systems; an eighth system is achieved through a change in system settings. The largest of these systems is a fully nested AChem that contains all the AChem and linking nodes used in our systems, Figure 18.

We combine the systems with two graph fragments, labeled Parameter Setting and Transfers in Figure 18. These provide a means of communication between the two systems. The five stages shown in Figure 18 are:

Initialization: Initial tanks of JA-AChem particles and initial swarm agents are loaded into the system and stored separately with matching indexing to allow for reference between the two.

Parameter setting: Parameter values are generated for each SwarmChem agent, based on the particles in its associated JA-AChem tank. The parameter values are pushed to the environmental container **V:**parameters (Figure 18). The swarm then updates itself by reading these values.

SwarmChem: The SwarmChem particles are updated and moved using a single SwarmChem time step.

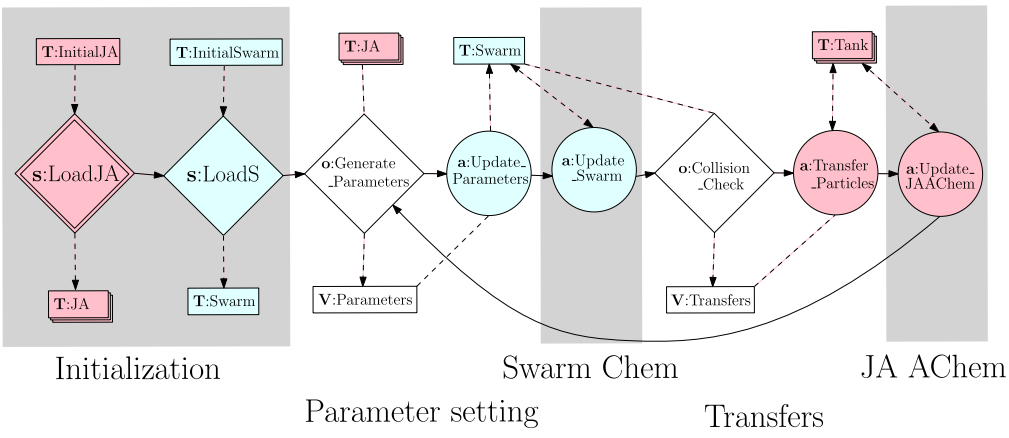

Figure 18. Macro-level NestedChem graph in MetaChem. JA-AChem nodes are shown in pink, SwarmChem nodes in blue. White nodes are either shared or not natively part of either AChem.

Transfer: SwarmChem assesses whether any collisions have occurred between its particles. It pushes a record of these collisions to the environmental container **V:**transfers. JA-AChem reads this container, and uses the results to exchange particles between tanks, based on the SwarmChem collisions.

JA-AChem: The JA-AChem updates by performing a number of bonding and decomposition attempts. All tanks are independent mass-conserving well-mixed tanks.

There are apparently four invalid edges in the macro system graph of NestedChem (Figure 18): (**a:**Update_Parameters, **T:**Swarm), (**a:**Swarm_Update, **T:**Swarm), (**a:**Transfer_Particles, **T:**Tank), and (**a:**JA-AChem_Update, **T:**Tank). All of these edges appear to allow actions to push to tanks, which is not allowed (Table 3). In the case of **a:**Swarm_Update and **a:**JA-AChem_Update we have seen the expanded graphs of these nodes in the macro graphs of each system, Figures 10 and 14. In those graphs we see that the actions carried out by these nodes mean they always move the particles to samples before making any changes. Here we connect directly to the tanks; as this is a macro graph, a form of pseudocode, this is actually implemented with these nodes expanded through the macro graphs shown previously down to the micro graph (Figures 11 and 15). At these levels of description the content of these tanks is moved to samples before being used.

In the case of **a:**Transfer_Particles, if we were to expand this node, we would see that all of its operations are carried out by samplers, and there is therefore no difficulty that before starting the particles have not been moved to a sample.

Finally, in the case of **a:**Update_Parameters the process function is applied over all particles in the system, meaning the sample would be the entire tank, so—in another abuse of notation and to avoid introducing two further control nodes and a container to move the entire contents back and forth—we allow the node to connect directly to the tank. It should be assumed that in the expanded form the necessary sampling would occur.

8.3.1 Modular Systems

From this full system we can derive eight variant systems. The control flow of these systems is shown in Figure 19.

I. *Nested.* The full nested AChem system as shown in Figure 18.

II. *Nested without collision.* JA-AChem particles are not transferred between tanks, but still determine the parameter values of agents in the SwarmChem.

III. *SwarmChem.* SwarmChem agents randomly exchange parameter values on collision; there is no communication with the JA-AChem.

IV. *SwarmChem without collision.* A very basic form of SwarmChem in which the agents interact only through boidlike flocking behaviors.

V. *JA-AChem single tank.* A single well-mixed tank of JA-AChem. The same number of evaluations are used per generation, and the same number of starting particles are also used as in the other systems.

VI. *JA-AChem multiple tanks with no interaction.* A JA-AChem with the same number of tanks as in the nested version; there are fewer atoms and particles in each tank, but the same number of overall atoms and evaluations are used.

VII. *JA-AChem multiple tanks with random transfers.* The same system as in VI, but with tanks randomly selected to randomly transfer particles between them.

VIII. *JA-AChem multiple tanks with grid transfers.* The same as in VII, but transfer tanks selected based on a Moore neighborhood.

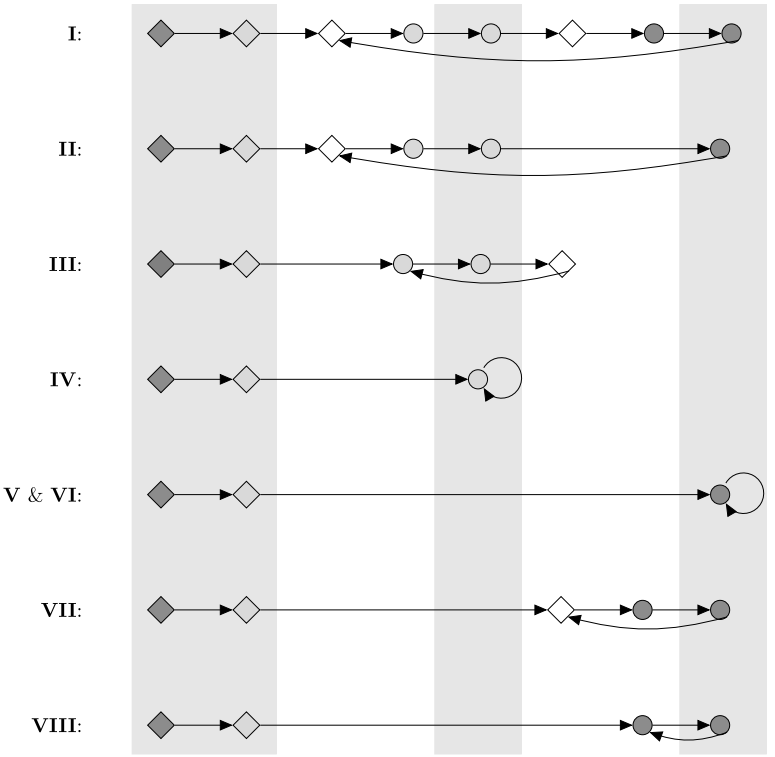

Figure 19. Various combinations of JA-AChem and SwarmChem. See text for details.

8.3.2 Discussion

In the JA-AChem level of the systems the resultant number of particles in the tanks should quickly stabilize, but we expect the systems with transfers to be less stable than others. Particles being transferred in and out of the tanks should disturb any equilibrium.

We also expect to see larger particles in the partitioned systems, as the smaller size of the tanks limits the sampling possibilities, increasing the chances of selecting molecules that already contain multiple particles. As these are used and the number of particles in the tank decreases, these probabilities should further increase.

We can observe many different statistics on the agents of the swarm. In homogeneous flocking the relative position of an agent to its visible neighbors should be very similar across agents, as a flock all have the same perception radius and tendency for avoidance. In SwarmChem these have greater variation but should be similar in sets of agents forming a swarm. Here we expect to see greater variation in the nested SwarmChem, where all values of the perception radius and the tendency for avoidance are possible.

Results, analysis, and further discussion of these nested systems can be found in [17, 18].

9 Conclusion and Future Work

9.1 Conclusion

We now have a formal language in which to discuss different AChems. In MetaChem, we can readily take the same "chemicals" and investigate their behavior in different "glassware," separating out the

contribution of the underlying low-level bonding rules from the environmental effects of how the particles are brought together.

All current systems we are aware of in the literature can be described by Static Graph MetaChem.

To show the power of this modularization and graph-based representation, we have presented two case studies of AChems. The first of these is our own chemistry JA-AChem, originally based on algebraic structures, but here re-described in the MetaChem format. The second AChem is Swarm Chemistry, chosen for being a well-established AChem that is not well described in the (S, R, A) format. SwarmChem and JA-AChem are very different AChems, with next to no overlap in their nature. SwarmChem also represents an independent example of description of an existing AChem in MetaChem.

We combine these using a graph structure that is widely applicable for the joining of two artificial chemistries. This is one possible use of Static Graph MetaChem; there are others, as well as extensions of MetaChem, described below.

9.2 Future Work

9.2.1 Other Combinations of AChems

We have illustrated one approach for combining artificial chemistries here, with two specific AChems, but the potential is much broader. New and different hierarchical AChem combinations could be tried. There are also mixed systems and joint systems to consider. Mixed systems would share tanks or spatial environment, with a new interaction added between the different types of particles. Joint systems would work with a combined particle made up of a particle from each system.

9.2.2 Static Graph MetaChem: Reuse and Toolsets

The Static Graph MetaChem described here is a first step towards a standardized framework for AChems. It is not just a mathematical framework; it can be and has been implemented in software [19]. These graphs provide more than a simple visualization: They are a new way to design, implement, and run AChems.

MetaChem provides a move forward in designing AChems. As we gather more descriptions in this framework, designers can begin to make use of parts of existing descriptions in new systems. Additionally, we can start to standardize output values from AChems, and make use of standard visualizations and reporting of results. The use of modularized structured nodes with defined functions should allow designers to define new nodes easily.

We have defined a general method of composition using indirect communication (a form of environment orientation [10]) with macro-level graphs. The ability to join systems and use modularity to share parts of algorithms could provide, after more development, significant speedups in designing and implementing new AChems.

The modularity and clear designation of particles and environmental properties allows the design of generic analysis tools, visualization tools, and metrics that can be used across similar systems. An AChem can provide a set of particles and their positions to a visualizer, regardless of the system's other properties. A more general-purpose proximity-based analysis for higher-level object identification, such as that used in more recent SwarmChem work [26, 27], becomes reusable.

9.2.3 Dynamic MetaChem

Above we describe *Static Graph* MetaChem. The graph exists before the system is run and does not change at run time, similar to most programs. A static graph version of MetaChem could be defined with a set of graph-rewriting rules that generate the graph. The rule set could produce a particular graph or multiple possible graphs, such as the ones in Figure 19.

This use of graph-rewriting rules then provides a natural way to make the topology dynamic, during AChem execution. A system could grow at run time, and could grow differently according to differences in the produced particles and variables.

A *dynamic edge graph* MetaChem would allow the graph to add and remove edges during run time. This could use a further type of control node to be responsible for this rewrite. In terms of the hierarchy, Figure 1, these nodes would fall under the grouping of control flow admin nodes. Such a system could reorder its own control flow, or even connect entirely new nodes or subgraphs to the control flow that, while existing at the start, were not connected.

We could use these new control nodes to trigger events in the system according to specific conditions, such as complexity. We could also use this as part of evolving artificial chemistries, by having a set of nodes to start with and allowing edges to change over time until the control flow becomes stable. This could be controlled by the system itself, so it would "learn" an artificial chemistry.

9.2.4 Evolving MetaChem

For true evolution and change we would move to full *graph language* MetaChem (or *dynamic graph* MetaChem), which could create and destroy its own nodes and edges at run time. The graph could grow, and could remove parts that were no longer needed, allowing it to prune its own process. This type of system would allow the AChem to change completely at run time, so it could truly transition and change abstraction levels and experiments as it ran. This could enable paths in open-ended evolution and open-ended systems research.

If we allow the set of particles to be the graph *rules* that generate the graph, and the reactions change those rules, then we can evolve how graphs form. This would allow systems capable not only of changing at run time, but of changing their basic components and how they can run during their execution. With suitable initial rule sets and reactions this could allow for the production of completely unexpected AChems with as little design bias as possible.

This could allow the design and growth of a system capable of self-reflection and change at run time. This in turn opens new possibilities for transitions towards open-ended evolution [30], as we can build systems capable of reacting to new emergent objects or behaviors if they can be identified. For example, if a system identified a set of objects within itself, it could then attempt to model those objects at a higher level and improve that model with information from the original low-level implementation [16].

9.2.5 MetaChem as an AChem

We can consider MetaChem itself as an AChem, where the atoms are graph nodes, the links are graph edges, and the composite particles are (potential) AChems. We can consider an isolated MetaChem subgraph as forming a sub-AChem or a full AChem. An instance of MetaChem is therefore not a single graph but a collection of graphs. So MetaChem provides both the language to describe AChems, and a process to build, compose, and evolve AChems. This is a key advantage of using a graph-based formal framework in the definition of MetaChem.

Acknowledgments

The research for this work was done with Ph.D. funding from the Department of Chemistry, University of York, UK.

References

1. Banzhaf, W., & Yamamoto, L. (2015). *Artificial chemistries*. Cambridge, MA: MIT Press.
2. Dittrich, P., Ziegler, J., & Banzhaf, W. (2001). Artificial Chemistries—A review. *Artificial Life*, *7*(3), 225–275.
3. Faulconbridge, A. (2011). *RBN-World: Sub-symbolic artificial chemistry for artificial life*. Ph.D. thesis, Department of Biology, University of York.
4. Faulconbridge, A., Stepney, S., Miller, J. F., & Caves, L. S. D. (2011). RBN-World: A sub-symbolic artificial chemistry. In G. Kampis, I. Karsai, & E. Szathmáry (Eds.), *Proceedings of ECAL 2009, Budapest, Hungary* (pp. 377–384). New York: Springer.

5. Faulkner, P., Krastev, M., Sebald, A., & Stepney, S. (2018). Sub-symbolic artificial chemistries. In S. Stepney & A. Adamatzky (Eds.), *Inspired by nature* (pp. 287–322). New York: Springer.
6. Faulkner, P., Sebald, A., & Stepney, S. (2016). Jordan algebra AChems: Exploiting mathematical richness for open ended design. In C. Gershenson et al. (Eds.), *Proceedings of the Artificial Life Conference 2016* (pp. 582–589). Cambridge, MA: MIT Press.
7. Faulkner, P., Sebald, A., & Stepney, S. (2017). Tuning Jordan algebra artificial chemistries with probability spawning functions. In C. Knibbe et al. (Eds.), *Proceedings of ECAL 2017, Lyon, France* (pp. 497–504). Cambridge, MA: MIT Press.
8. Grimm, V., & Railsback, S. F. (2005). *Individual-based modeling and ecology*. Princeton, NJ: Princeton University Press.
9. Hayes-Roth, B. (1985). A blackboard architecture for control. *Artificial Intelligence*, *26*(3), 251–321.
10. Hoverd, T., & Stepney, S. (2015). Environment orientation: A structured simulation approach for agent-based complex systems. *Natural Computing*, *14*(1), 83–97.
11. Hutton, T. J. (2007). Evolvable self-reproducing cells in a two-dimensional artificial chemistry. *Artificial Life*, *13*(1), 11–30.
12. Krastev, M., Sebald, A., & Stepney, S. (2016). Emergent bonding properties in the spiky RBN AChem. In C. Gershenson et al. (Eds.), *Proceedings of the Artificial Life Conference 2016* (pp. 600–607). Cambridge, MA: MIT Press.
13. Liu, Y. (2018). The artificial ecosystem: Number soup (part II). *arXiv preprint arXiv:1801.04916*.
14. Madina, D., Ono, N., & Ikegami, T. (2003). Cellular evolution in a 3D lattice artificial chemistry. In W. Banzhaf et al. (Eds.), *Proceedings of ECAL 2003* (pp. 59–68). New York: Springer.
15. McCrimmon, K. (2006). *A taste of Jordan algebras*. New York: Springer Science & Business Media.
16. Nellis, A., & Stepney, S. (2010). Automatically moving between levels in artificial chemistries. In H. Fellermann et al. (Eds.), *Proceedings of ALife XII, Odense, Denmark* (pp. 269–276). Cambridge, MA: MIT Press.
17. Rainford, P. F. (2018). *Algebraic approaches to artificial chemistries*. Ph.D. thesis, Department of Chemistry, University of York.
18. Rainford, P. F., Sebald, A., & Stepney, S. (2018). Modular combinations of artificial chemistries. In T. Ikegami et al. (Eds.), *Proceedings of the Artificial Life Conference 2018* (pp. 361–367). Cambridge, MA: MIT Press.
19. Rainford, P. F., Sebald, A., & Stepney, S. (2019). An object oriented implementation of the MetaChem framework. In H. Fellermann et al. (Eds.), *Proceedings of the Artificial Life Conference 2019* (pp. 119–126). Cambridge, MA: MIT Press.
20. Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. In M. C. Stone (Ed.), *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '87* (pp. 25–34). New York: ACM.
21. Sayama, H. (2009). Swarm chemistry. *Artificial Life*, *15*(1), 105–114.
22. Sayama, H. (2010). Robust morphogenesis of robotic swarms [application notes]. *IEEE Computational Intelligence Magazine*, *5*(3), 43–49.
23. Sayama, H. (2010). Swarm chemistry evolving. In H. Fellermann et al. (Eds.), *Proceedings of ALife XII, Odense, Denmark* (pp. 32–33). Cambridge, MA: MIT Press.
24. Sayama, H. (2011). Seeking open-ended evolution in swarm chemistry. In *2011 IEEE Symposium on Artificial Life (ALIFE)* (pp. 186–193). New York: IEEE.
25. Sayama, H. (2012). Evolutionary swarm chemistry in three dimensions. In C. Adami (Ed.), *Proceedings of Artificial Life 13* (pp. 576–577). Cambridge, MA: MIT Press.
26. Sayama, H. (2018). Seeking open-ended evolution in swarm chemistry II: Analyzing long-term dynamics via automated object harvesting. In T. Ikegami et al. (Eds.), *Proceedings of the Artificial Life Conference 2018* (pp. 59–66). Cambridge, MA: MIT Press.
27. Sayama, H. (2019). Cardinality leap for open-ended evolution: Theoretical consideration and demonstration by "hash chemistry." *Artificial Life*, *25*(2), 104–116.

28. Sayama, H. (2019). Complexity, development, and evolution in morphogenetic collective systems. In G. Georgiev, J. M. Smart, C. L. F. Martinez, & M. E. Price (Eds.), *Evolution, development and complexity: Multiscale evolutionary models of complex adaptive systems* (pp. 293–305). New York: Springer.

29. Soula, H. A. (2016). Generalized stochastic simulation algorithm for artificial chemistry. In C. Gershenson et al. (Eds.), *Proceedings of the Artificial Life Conference 2016* (pp. 590–597). Cambridge, MA: MIT Press.

30. Stepney, S., & Hoverd, T. (2011). Reflecting on open-ended evolution. In T. Lanaerts et al. (Eds.), *Proceedings of ECAL 2011, Paris, France* (pp. 781–788). Cambridge, MA: MIT Press.

31. Stepney, S., Polack, F., & Toyn, I. (2003). Patterns to guide practical refactoring: Examples targetting promotion in Z. In D. Bert, J. P. Bowen, S. King, & M. Walden (Eds.), *ZB2003: Third International Conference of B and Z Users, Turku, Finland* (pp. 20–39). Cham, Switzerland: Springer.

32. Watson, I., Sebald, A., & Stepney, S. (2019). A meta-atom based sub-symbolic artificial chemistry. In *Artificial Life Conference 2019* (pp. 127–134). Cambridge, MA: MIT Press.

Appendix: Mathematical Formalism

We provide a mathematical formalism of our system here. First we describe the static elements that make up the static graphs. Then we define the dynamic system state. Finally, we define the state transition function over the graphs that are used to capture the dynamics of a specific AChem.

A.1 Static Graph MetaChem

We have two static graphs capturing the control flow and information flow. We build up the definition as follows.

A.1.1 Nodes

The set of graph nodes is N. As shown in our hierarchy (Figure 1), our system is composed of container and control nodes. The graph nodes are partitioned into two sets: control nodes C and container nodes B:

$$\langle B, C \rangle \text{ partition } N \tag{15}$$

where the notation $\langle X_1, \ldots, X_n \rangle$ **partition** X means that the set X is partitioned by the n subsets X_i.

Each of the sets B and C is partitioned further.

The container nodes B comprise three categories: environment nodes V, tank nodes T, and sample nodes S:

$$\langle V, T, S \rangle \text{ partition } B \tag{16}$$

Control nodes are more complicated in the hierarchy, but the static components partition the set into action (C_a), decision (C_d), observer (C_o), sampler (C_s), and termination (C_t) nodes:

$$\langle C_a, C_d, C_o, C_s, C_t \rangle \text{ partition } C \tag{17}$$

A.1.2 Edges

Our hierarchy (Figure 1) also contains control flow and information flow. These appear in our static graphs as edges. We define an edge as a pair of nodes. Edges are either control edges or information edges, E_G and E_I.

Control edges: These are edges between control nodes:

$$E_G \subseteq C \times C \tag{18}$$

Different subtypes of control nodes can have different numbers of exiting edges. Define **target** to map a source control node to the set of target control nodes connected to it by an edge in E_G:

$$\textbf{target} : C \to \mathbb{P}C \tag{19}$$

$$\forall c : C \mid \textbf{target}(c) = \{c_s : C \mid (c, c_t) \in E_G\} \tag{20}$$

Decision control nodes have multiple targets; all other control nodes have a unique target:

$$\forall c : C_d \mid \#\textbf{target}(c) > 1 \tag{21}$$

$$\forall c : C \setminus C_d \mid \#\textbf{target}(c) = 1 \tag{22}$$

Information edges: These come in three varieties.

Read edges, E_{read}, are directed from control nodes to containers, and indicate which containers' information a control node can read. In the graphical notation they are shown as undirected edges, as there is no change to the container node:

$$E_{\text{read}} \subseteq C \times B \tag{23}$$

Pull edges, E_{pull}, are directed from containers to control nodes, and indicate the containers that a control node can remove information or objects from. Every E_{pull} edge must have a corresponding E_{read} edge:

$$E_{\text{pull}} \subseteq B \times C, \quad E_{\text{pull}} \subseteq E_{\text{read}}^{-1} \tag{24}$$

Push edges, E_{push}, are directed from control nodes to containers, and indicate the containers that a control node can push information and objects to. Every E_{push} edge must have a corresponding E_{read} edge:

$$E_{\text{push}} \subseteq C \times B, \quad E_{\text{push}} \subseteq E_{\text{read}} \tag{25}$$

There are limits on the containers that different node types are allowed to have pull and push edges with; see Table 4.

A.1.3 Graphs

We have two graphs, G_Ω, I_Ω, of our system Ω, capturing control and information respectively. For the purpose of this definition we assume we are always referring to elements of a given system, and so drop the system label, so that our graphs G_Ω and I_Ω become G and I:

$$G = (N, E_G), \quad I = (N, E_I) \tag{26}$$

Table 4. Container types that control nodes are allowed to have push and pull edges with.

	Action	Decision	Sampler	Observer	Termination
Tank *T*			✓		
Sample *S*	✓		✓		
Variable *V*	✓			✓	

A.2 Dynamic System State

Now that we have these static graphs, we can start a dynamic process guided by them. We denote the dynamic aspects of our system using the Greek alphabet, to distinguish them from static components.

Container nodes B can contain particles of type Φ. The structure of the set Φ of particle types is application-dependent. We define the contents P of such a node as a bag (multiset) of particle types:

$$P = \Phi \to \mathbb{N} \tag{27}$$

where $\mathbb{N}$ is the set of natural numbers counting how many instances of each particle type there are in the bag. Here we refer to the content of a container using the mapping from container to particle bag; in the informal sections above we abuse the notation and refer to the content of containers simply by the container label, for readability and brevity.

Environment nodes V contain environment information of type Ψ. We do not here further specify the structure of the set Ψ; it is application-dependent.

The current state node $c \in C$, a pointer in this static graph case, is a control node dynamically assigned and changing over time. This pointer indicates the current control node, whose transition function is to be run in order to find the next state of the system.

The full system state comprises five components: the control graph G, the information graph I, the current state node $c \in C$, a mapping from the container nodes to the bag of particles they contain ($\phi \in B \to P$), and a mapping from the environment nodes to the dynamic environment information they contain ($\nu \in V \to \Psi$). The system being formalized here has static graphs G and I, so we do not here include them in the system state, rather taking them to be globally defined. The set of system states is

$$\Omega_s = C \times (B \to P) \times (V \to \Psi) \tag{28}$$

We define a specific system state ω as the triple $(c, \phi, \psi) \in \Omega_s$. The initial state of the system has $c = c_0$, the identified start node.

A.3 Transition Functions

Each node has a transition function of the whole system state, as nodes can access and affect neighboring nodes:[5]

$$\delta : \Omega_s \to \Omega_s \tag{29}$$

5 It would be possible to define local state transition functions, and "promote" them [31] to a global state transition function, but the mathematical machinery needed to do so is here more cumbersome than a direct definition.

The overall transition function is decomposed into the component functions: *read*(), *check*(), *pull*(), *process*(), *push*(), *next*(). For any node some of these may be null (identity) functions. For some kinds of nodes, some components are always null or defaulted (Table 2).

Using ; to indicate strict ordering of application of functions from left to right gives the following definition of δ:

$$\delta = \mathit{pull}; \mathit{process}; \mathit{push}; \mathit{check}; \mathit{read}; \mathit{next} \tag{30}$$

Each of these transition function components plays a different role in the transition and uses a different aspect of the state. These functions are summarized in Figure 9 and formalized below.

The transition function components exploit a local state, which exists only for the duration of the transition. This comprises labeled bags of particles (labeled by their source container node), and labeled environment variables (labeled by their source environment node):

$$\mathit{Local} = (B \to P) \times (V \to \Psi) \tag{31}$$

We define a specific local state as the pair $(\phi_l, v_l) \in \mathit{Local}$.

The local state disappears as soon as the transition function is completed, so control nodes have no lasting state or memory. Any information used by a control node must come from containers at the start of a transition, using the *read*() or *pull*() function, and any information or objects that should remain in the system should be written back to a container by the *push*() function.

A.3.1 *read*()

The *read*() function allows a node to collect information from external containers into the temporary local containers, where it can to be used by the following transition functions. This does not modify the system state:

$$\mathit{read} : \Omega_s \to (\Omega_s \times \mathit{Local}) \tag{32}$$

$$\mathit{read}(c, \phi, \psi) = ((c, \phi, \psi), (\phi_l, \psi_l)) \tag{33}$$

The containers and environment nodes from which the current node c can read are the ones attached by read edges:

$$B_c = \{b \in B \mid (c, b) \in E_{\text{read}}\} \tag{34}$$

$$V_c = \{v \in V \mid (c, v) \in E_{\text{read}}\} \tag{35}$$

The default behavior of *read*() is to copy all the readable containers to the local state:

$$\phi_l = \{(b, \rho) \mid b \in B_c \wedge (b, \rho) \in \phi\} \tag{36}$$

$$\psi_l = \{(v, p) \mid v \in V_c \wedge (v, p) \in \psi\} \tag{37}$$

In practice, implementations may choose to read only a subset of the information in the readable containers.

A.3.2 *check*()

The *check*() function uses the local information to generate a threshold probability, which is used to determine whether the rest of the transition (the part that actually alters containers) occurs, or the process exits at this point. This packages any probabilistic aspects of the execution of the transition.

The check function uses a probability spawning function *psf* [7] to determine if the rest of the transition will occur. The default behavior in this case is to return True, which it does for administrative nodes, which always operate in a deterministic manner.

The execution checks the generated probability $psf(\phi_l, \psi_l)$ against a uniform random number, r. If our threshold probability is less than r, we continue; otherwise we exit:

$$\begin{cases} \delta = next, & psf(\phi_l, \psi_l) < r \\ check = Id(\Omega_s \times Local) & \text{otherwise} \end{cases} \tag{38}$$

where $r \in [0 : 1]$ is a uniformly distributed random number.

Either the transition function exits and does not proceed to any further functions, else the other functions are executed as expected. In either case, *check* makes no change to the state of the system, or the local state.

A.3.3 *pull*()

The *pull*() function removes information from connected containers. Any information removed has (potentially) been copied to the local state in *read*(), where it is available for local processing. This modifies the system state, but not the local state:

$$pull : (\Omega_s \times Local) \to (\Omega_s \times Local) \tag{39}$$

$$pull((c, \phi, \psi), (\phi_l, \psi_l)) = \left(\left(c, \phi', \psi'\right), (\phi_l, \psi_l)\right) \tag{40}$$

The containers and environment nodes from which the current node c can pull are the ones attached by pull edges:

$$B_c = \left\{b \in B \,|\, (b, c) \in E_{\text{pull}}\right\} \tag{41}$$

$$V_c = \left\{v \in V \,|\, (v, c) \in E_{\text{pull}}\right\} \tag{42}$$

The pull function may only change containers connected by pull edges, and then only to delete information from them:[6]

$$\forall b \in B_c \mid \phi'(b) \textsf{ subbag } \phi(\mathbf{b}) \tag{43}$$

$$\forall b \in B \setminus B_c \mid \phi'(b) = \phi(b) \tag{44}$$

$$\forall v \in V_c \mid \psi'(v) \textsf{ subenv } \psi(\mathbf{v}) \tag{45}$$

$$\forall v \in V \setminus V_c \mid \psi'(v) = \psi(v) \tag{46}$$

The default behavior of *pull* is to do nothing: $pull = Id(\Omega_s \times Local)$.

A.3.4 *process()*

The *process*() function acts as the main computation for the node. It modifies the local state of particles and variables, including creating new particles and variables and destroying old ones. It does not modify the system state:

$$process : (\Omega_s \times Local) \to (\Omega_s \times Local) \tag{47}$$

$$process((c, \phi, \psi), (\phi_l, \psi_l)) = \left((c, \phi, \psi), \left(\phi_l', \psi_l'\right)\right) \tag{48}$$

This function is entirely application-dependent. When c is a decision node, $c \in C_d$, the output *Local* state shall contain information to determine the choice of the next node.

A.3.5 *push()*

The *push*() function adds information from the local state to connected containers. This modifies the system state and preserves the local state:

$$push : (\Omega_s \times Local) \to (\Omega_s \times Local) \tag{49}$$

$$push((c, \phi, \psi), (\phi_l, \psi_l)) = \left(\left(c, \phi', \psi'\right), (\phi_l, \psi_l)\right) \tag{50}$$

The containers and environment nodes to which the current node c can push are the ones attached by push edges:

$$B_c = \left\{b \in B \mid (c, b) \in E_{\text{push}}\right\} \tag{51}$$

$$V_c = \left\{v \in V \mid (c, v) \in E_{\text{push}}\right\} \tag{52}$$

6 subbag has the obvious definition: There may not be more particles of any given type after than before. subenv is application-dependent, but should conform to the idea of removing information.

The push function may only change containers connected by push edges, and then only to add a new object to the container[7] with information from the local state:

$$\forall b \in B_c \mid \phi'(b) \ \textbf{combinedwith} \ \phi(\mathbf{b}) \tag{53}$$

$$\forall b \in B \setminus B_c \mid \phi'(b) = \phi(b) \tag{54}$$

$$\forall v \in V_c \mid \psi'(v) \ \textbf{combinedwith} \ \psi(\mathbf{v}) \tag{55}$$

$$\forall v \in V \setminus V_c \mid \psi'(v) = \psi(v) \tag{56}$$

The default behavior of *push* is to do nothing: $push(\omega, (\phi_b, \psi_l)) = (\omega, (\phi_b, \psi_l))$.

A.3.6 *next()*

The *next*() function moves the control pointer to the next node and destroys the local state. This modifies the pointer node component of the system state:

$$next : (\Omega_s \times Local) \to \Omega_s \tag{57}$$

$$next((c, \phi, \psi), (\phi_l, \psi_l)) = (c', \phi, \psi) \tag{58}$$

The nodes to which the current node pointer c can move to are defined by the control edge(s) from the current node:

$$c' \in \textbf{target}(c) \tag{59}$$

This set is a singleton set, except for decision nodes. For decision nodes, the choice of which element to go to next is provided in the *Local* information.

7 If an add is performed on a container to add a variable that already exists, the behavior is undefined and implementation-dependent; updating is therefore done by pulling the variable to remove it and then pushing it to re-add it.