An Object Oriented Implementation of the MetaChem framework

Penelope Faulkner Rainford[1,3,4], Angelika Sebald[1,4] and Susan Stepney[2,4]

[1]Department of Chemistry, University of York, UK
[2]Department of Computer Science, University of York, UK
[3]Department of Mathematical Sciences, Durham University, UK
[4]York Cross-disciplinary Centre for Systems Analysis
penelope.s.rainford@durham.ac.uk

Abstract

Our MetaChem framework supports the definition and combination of artificial chemistries. Here we describe an implementation of MetaChem in an object oriented language. We briefly define MetaChem, and provide an example in the form of a toy AChem: StringCatChem. We present the class hierarchy used to define MetaChem such that the implementation can run directly from a graph description of some AChem. This matches the description given by the formal framework definition. We also describe some generic functions of MetaChem that have been implemented and used in StringCatChem. This implementation is available on GitHub.

Introduction

Artificial chemistries (AChems), like other areas of artificial life, are predominantly software based. Mathematical models of our systems provide rigour in our definitions, yet they must also be implementable. We have developed MetaChem, a mathematically rigorous framework for defining single AChems, and for compositing different AChems into larger systems; it aims to provide a unified description language for all AChems. The formal MetaChem framework is defined in (Rainford, 2018); here we give a brief summary.

Our framework is designed to replace the only earlier framework for describing AChems presented in (Dittrich et al., 2001). That system is not designed with implementation in mind; rather, it is a tool to help describe AChems and draw comparisons. It splits an AChem definition into three parts, using the triplet (S, R, A): a set of particles S, rules for reactions R, and the algorithm A. All non-particle aspects of a system are combined in A as part of the algorithm, including spatiality, rule application, global variables, timing, and logging.

That framework is helpful and complete for describing the well-mixed tank-based systems of early work, but since its introduction AChems have changed considerably. For example, many systems have spatial elements (Ono and Ikegami, 2001; Hutton, 2007). There are subsymbolic chemistries where particles have internal structure (Faulconbridge, 2011; Faulkner et al., 2018), and automata chemistries with instruction sets packaged with processors for particles (Hickinbotham et al., 2010; Ofria and Wilke, 2004). There are even AChems without direct interaction between particles (Sayama, 2011). These new AChems pose difficult questions for the old framework: is position an aspect of the particle S or the algorithm A? Is the processor in S or A? If we have no physical linking, what is in R?

Our new framework encompasses this increased complexity by considering the entire system as a sequence of events occurring on objects in an environment, rather than as a function of rule application to particles. With this we are able to better describe and compare the diverse set of AChems.

A generic description language such as this should also aim to make implementation easier. The ability to build tools that work for multiple systems is one of the primary goals for making these AChems compatible. This means that we need to be able to implement the framework in order to fully leverage its potential power. Previous work on implementation of AChem systems has been done (Bersini, 1999). However that work focused on a single AChem and does not generalise.

We have implemented our MetaChem framework and used it to combine two quite dissimilar AChems into a single system (Rainford et al., 2018a). In this paper we focus on implementation issues: how to convert the formal mathematical framework into a generalised reusable object-oriented implementation. We illustrate and examine the implementation of MetaChem here using StringCatChem, a toy AChem based on string concatenation. A Python implementation of the MetaChem framework, including StringCatChem, is available at github.com/faulknerrainford/MetaChem.git

A key requirement is that our implementation matches our framework as closely as possible. This makes it easy for different users to translate from one format to the other. Additionally, we want our implementation of the high level MetaChem structure to be independent of the implementation of the lower level AChem-specific particles and algorithms, to allow rapid integration.

	Primary Focus	Auxiliaries
Objects:	Particles	Variables
Containers:	Tanks	Environment

Table 1: Common parts of AChem Systems

MetaChem

Modularising Artificial Chemistries

There are axiomatic concepts in all AChems that we build on. We work on the basis of small components interacting to generate our systems. We are interested in the emergent properties and behaviours of these systems. To differentiate an AChem from an Individual Based Model we add requirements for simplicity and tractability in our particles and their interactions. The intention is that these systems work over large collections of individuals over long time periods, though most are currently limited by computational capability. To consider computational issues in our models we need our frameworks and models to consider implementation.

From these axiomatic concepts we identify many common elements of AChem systems. We use these as the basis for a bottom-up approach to systematic modularisation of AChems. Small, simple individuals and their interactions are our primary focus. We call these individuals *particles*. These exist in all AChem systems. Systems also have other variables, properties and values; we describe these in the *environment*. Much like in real chemistry, we separate the description of the "glassware" from our consideration of its particle contents. We have multiple *containers* in our system, which allow us to isolate particles and move them (analogous to the "beakers", "pipes", and "valves" comprising the "glassware"). This splits the dynamic parts of our system as shown in Table 1.

These components handle the "things" in our systems. 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 modularised 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, whilst using pre-existing elements to implement less unique aspects of their systems.

Having divided our "things", we define our control flow in relation to these divisions. We *modify particles*, similar to reactions and interactions in chemistry. We *record observations* of our system. We *modify the 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.

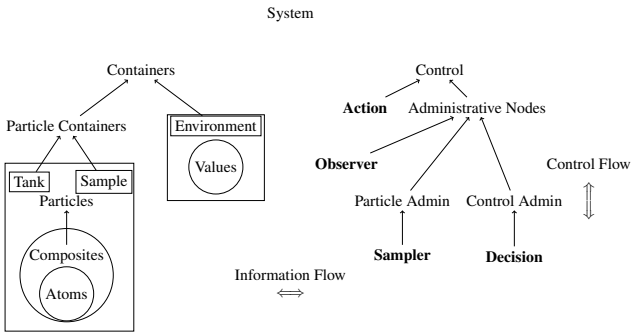


Figure 1: Conceptual structure of modularisation of AChems

We conceptualise these components 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 formalised as graph *nodes* (Containers, Control), and as graph *edges* (Information Flow, Control Flow).

Control items are static nodes in the graph: their location and connectivity defined at the start, and remains unchanged as the AChem executes. These control nodes are connected by Control Flow edges, which together define the system's specific algorithm.

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

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

We now describe these nodes and edges in sufficient detail to explain their implementation. For full formal definitions, see (Rainford, 2019).

Control nodes and Edges

The control flow of our system defines the AChem's algorithm. The control evaluates a node's definition, and moves to the next node. In the implementation, it iteratively executes a node's transition function, and moves on to the next node; by traversing the graph in this manner it performs the relevant computation.

All control nodes execute the same basic state transition: the state changes, then control moves to the next node. The overall transition is defined through five individual transition functions: *read*(), *check*(), *pull*(), *process*(), *push*(), executed sequentially:

$$transition = read \fatsemi check \fatsemi pull \fatsemi process \fatsemi push$$

where ⨟ indicates strict ordering of function application from left to right.

Each of these transition function components plays a different role in the transition and thus uses a different aspect of the state.

read(): the current node collects information from (connected) external containers into temporary local containers, for used by the remaining transition functions. This action does not modify the external containers in any way. One can think of it copying the read particles and values.

check() : the current node uses its local information to generate a threshold probability value p, which it uses to determine if the rest of the transition function (the part that actually alters containers) occurs. In the implementation, it generates a uniform random number r; if $p < r$, execution continues, otherwise it exits and moves to the next node.

pull(): remove particles and change information in external containers where appropriate. Any information so pulled must already been copied to local containers by *read*(), where it is available for local processing. Note that read information does not have to be pulled (that is, it can be copied, rather than moved).

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

push(): push variables and particles from the local variables into external containers; wipe the local containers' contents.

Transition functions operate on local state, which exists only for the duration of the transition. Local particle containers and local environment containers are destroyed as soon as the transition function is completed, so the 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*() or *pull*(); any information or objects that need to remain in the system are written back to a container by *push*().

These operations are summarised in Figure 2 and discussed in the context of specific node types below.

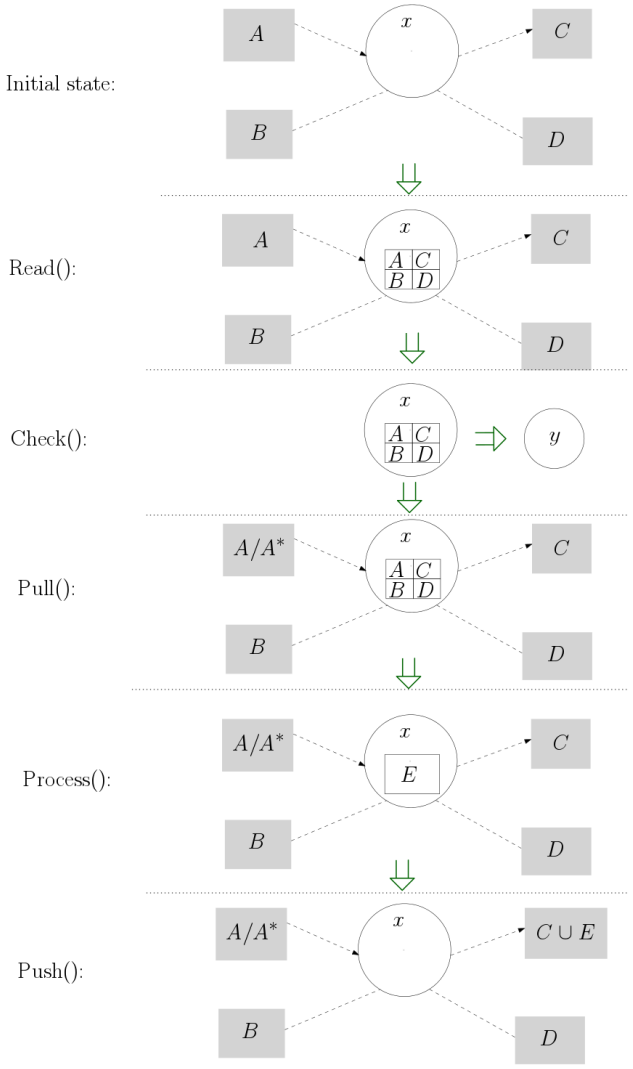


Figure 2: Summary of movement and processing of information done by transition functions in a node. A, B, C, D are container contents; $A^* \subseteq A$.

Control node subtypes

Our control nodes are partitioned into four subtypes: action, decision, sample and observer. We define these node subtypes by requiring some of the transition function parts to be null (identity), 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, process the particles in the system for the reaction to happen. It is not limited by which transition functions it executed, but it is limited by which containers it can *push*() to, see Table 3. The limit to modify only samples allows parallelisation, 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 nodes. It is limited to just *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 the information, and it does not modify any particles or environment variables. It therefore has only a *read*(), *pull*() and *push*() function.

	action	decision	sample	observer
read	✓	✓	✓	✓
check	✓			
pull	✓		✓	✓
process	✓	✓		✓
push	✓		✓	✓

Table 2: Transition functions used by different types of control nodes; unchecked functions always return their default behaviour

	action	decision	sample	observer
tank			✓	
sample	✓		✓	
environment	✓			✓

Table 3: The set of container nodes that can be modified (push()/add() or pull()/remove()) by a control node, read() can always be performed on any container

Observer : observe but do not modify particles; modify the environment. It has a *read*() to view containers, and a *pull*() to allow it to edit environment variables only. It has a *process*() that allows it to compute summary statistics and changes to the environmental variables, and a *push*() to commit those changes back to the environment.

Container nodes

Container nodes are partitioned into two subtypes: Particle nodes and Environment nodes.

Particle nodes : mappings that take the node and the state of the system, and returns the set 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 partitioning of all the particles in the system. There are two types of particle nodes: samples and tanks. Tanks are protected containers. Particles in tanks can be moved in and out but cannot be changed; any changes must be made over samples, so the designer has to decide what will be changing.

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

Environment nodes : similar to particle 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 to 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.

Use of mappings. It is key that containers work using mappings. Any container has three ways to access the information stored in them: *read*(), which returns a copy of all the information in the container; *remove*([particle list]), which removes all particles (or variables in the case of environment containers) in the list from the container (this is called by *pull*() during transitions so should be preceded by a *read*()); and *add*([particle list]), which adds the particles to the container (or adds a variable to the environment in the case of environment containers).

These mappings are a way to keep nodes independent of implementation. They define an interface. As long as the implementer can provide these three functions, they can store the items in what ever way they wish. This also means that control nodes that move objects and read them can be independent of the implementation of their storage. This allows for greater reuse of control nodes across systems.

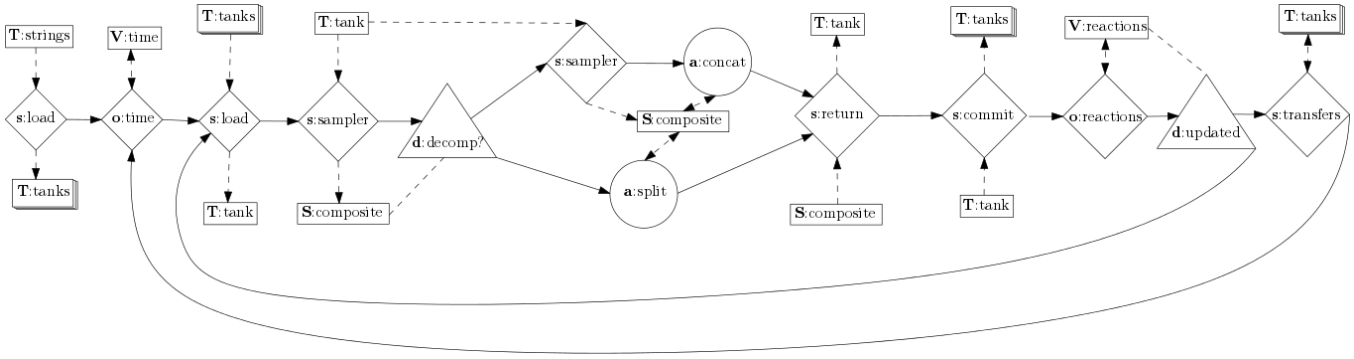
The limits on access to containers placed on control nodes is given in Table 3. The *read*() action is always possible for a control node of any container node (information is always knowable). However, we limit the modification of containers by nodes to make it easier to track activity in the system.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

Example: StringCatChem

To help illustrate the use and power of MetaChem we introduce StringCatChem, a toy AChem. Its atoms are characters, the standard 26 lower case letters of the English alphabet. Composite particles are strings formed via concatenation. StringCatChem is situated in a collection of well-mixed (aspatial) tanks. When particles combine or split they remain in the tank. When we select a string we check if it contains any adjacent repeated letters; if so we split it between them. If not we select a second string at random and concatenate the two. We also randomly transfer strings between tanks.

The simplicity of this system means StringCatChem will continue to run until everything has formed a small number of large particles in each tank, all with matching letters at the starts and ends of them and no internal repeated letters. After this the system will not change. StringCatChem is therefore not a good choice of AChem if one wishes to study open-endedness or the transition to life. However, it makes a good example of implementation: the whole system can be implemented with 4 container nodes and 13 control nodes. The graph representation (and code) of the system is given in Figure 3.

The graph is a formal definition of the system itself; it is not merely a helpful visualisation of it. The code below it is a different concrete syntax of the same graph: it is the textual form for input to our interpreter. This is a description of the




```
edges = [[Sload, Otime], [Otime, Ssamplertank], [Ssamplertank, Ssamplerstring],
         [Ssamplerstring, Ddecomp],
         # Ddecomp splits control
         [Ddecomp, Ssamplerstringdecomp],
         [Ddecomp, Asplit], [Ssamplerstringdecomp, Aconcat],
         # Control merges again at Sreturn
         [Asplit, Sreturn], [Aconcat, Sreturn],
         [Sreturn, Scommit], [Scommit, Oreaction]], [Oreaction, Dupdate],
         # Dupdate splits control again
         [Dupdate, Stransfers],
         [Dupdate, Ssamplertank],  # Loop to the start of reaction
         [Stransfers, Otime]]  # Loop to the start to the generation

```

Figure 3: Graph for StringCatChem in visual and code form. For a key to the graph node shapes, see Table 4.

graph in terms of edges as pairs of nodes. The only further information the interpreter needs to run the StringCatChem is a limit on the number of transitions, and the specific implementations of the relevant transition functions.

The MetaChem framework is based on transitions rather than generations, as this is the natural step component; the MetaChem system has no built-in knowledge of generations or number of reactions.

Implementation

Now that we have overviewed the components of our graph-based MetaChem, here we discuss the implementation of the framework in Python. The hierarchical framework (Figure 1) translates readily to a class diagram, Figure 4.

Control nodes

The basic control node is defined as a transition function (Listing 1), which uses the five sub-functions. The transition function itself is used by almost all the subclasses. We also provide default null versions (which immediately *return*) of all of these five functions for use as null behaviours.

Our subclass nodes then use the default function in the top-level class for those functions not used by that type of node. The remainder of the functions are overwritten with default functions that perform basic functionality for that type of sub-node. For example, in the subclasses the default behaviour for *read*() is to read all information from all connected containers.

```
def transition(self):
    self.read()
    if self.check() < random.random():
        self.pull()
        self.process()
        self.push()
    pass
```

Listing 1: Transition function as defined in ControlNode class

```
def transition(self):
    self.read()
    return self.process()
```

Listing 2: Transition function as defined in Decision class

Decision does overwrite *transition*(), Listing 2, as in the case of Decision the transition needs to return a value back to the graph handler so it can transition to the correct node.

The graph handler checks if the node is running is a Decision and expects it to return a value.

Container nodes

Our container subclasses are very similar to the ContainerNode superclass. As long as a subclass implementation of a container provides a *read*(), *add*() and *remove*() function, the actual storage method is not important at this level.

For example, for StringCatChem we implement a list-based container. Listing 3 shows the *initialiser* and *read*() functions for this class.

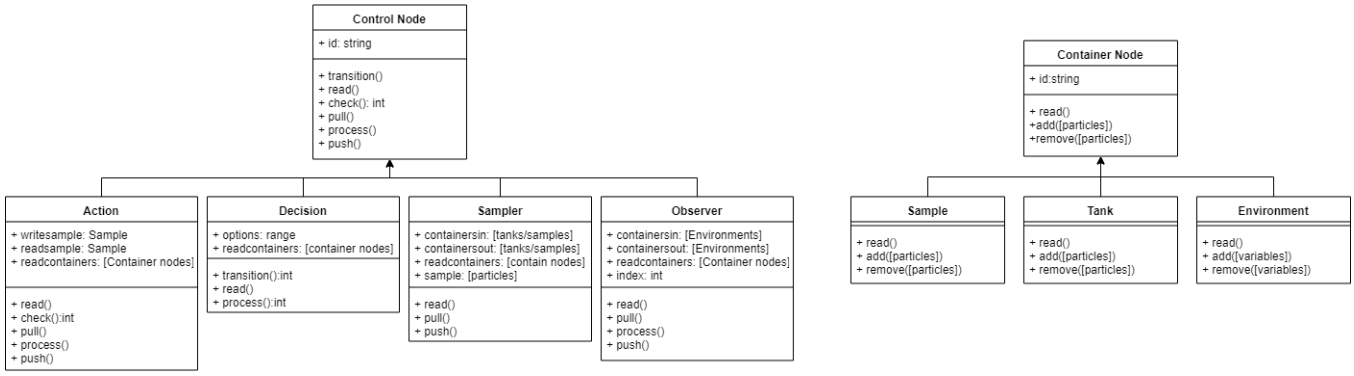


Figure 4: Class diagram of the implementation of MetaChem

```
class ListTank(node.Tank):

    def __init__(self):
        super(ListTank, self).__init__()
        self.list = None
        pass

    def read(self):
        return self.list[:]
```

Listing 3: ListTank class definition including initialiser and read function

We could replace this list with a dictionary. Or we could use a database, with *add*() adding new records and *remove*() deleting them or modifying them so they are not recorded as part of the tank. Or we could use an array or file instead. We can use any such implementation, as long as we can reduce the interactions with it to *read*(), *add*() and *remove*(). So we can choose an appropriate implementation for any particular system at a particular size without needing to have this affect our control nodes and algorithm.

Given our containers are all so similar, why do we implement these subclasses? We do this so that we can type-check containers to ensure the different types of control nodes are modifying only allowed nodes.

Generic common nodes

There are common nodes that we will use multiple times in a system and in many different systems. We implement such types of nodes in their own two modules, one for containers and one for control nodes.

One obvious such common node is a counter node, which increments a particular variable that is later used by a decision node to control looping. Such a counter and decision, implemented as ClockObserver and CounterDecision, are given in Listings 4 and 5. The ClockObserver requires that the *pull* and *push* actions modify the same value, and takes an amount by which the value is incremented. The CounterDecision is set to look at the same variable, and is initialised with a threshold that determines next node.

```
class ClockObserver(node.Observer):

    def __init__(self, containersin, containersout,
     readcontainers=None, increment=1):
        if containersin != containersout:
            raise ValueError("Clock must read and write
     to same variable")
        else:
            super(ClockObserver, self).__init__(
     containersin, containersout, readcontainers)
            self.increment = increment
            self.clock = 0
            self.variable = self.containersin
            pass

    def read(self):
        self.clock = self.variable.read()[0]
        pass

    def pull(self):
        self.variable.remove(self.clock)

    def process(self):
        self.clock = self.clock + self.increment
        pass

    def push(self):
        self.variable.add(self.clock)
        pass
```

Listing 4: ClockObserver node code

Both of these generic nodes are used in StringCatChem to track and respond to the number of reactions that have happened. Their use in StringCatChem is given in Listing 6. Here we loop until 200 reactions have occurred.

Edges

The graph module interprets our graphs, given in textual form, and runs them. It stores the set of nodes into a dictionary. Each node is a key; the value is the node or list of nodes to which it has outgoing edges. This allows the interpreter to move on to the next node. In most cases there is only a single control node as the value; in the case of decisions there is a list of control nodes.

Element	Description
Containers	
T	**Tank** containing particles.
S	**Sample** containing an editable subset of particles.
V	**Environment** containing non-particle variables and information in the system.
Control	
s	Information administration node moves particles between containers by **sampling** them
o	**Observes** particles and produces summary statistics (saved in the environment), also an information administration node.
d	Control administration node makes weighted **decisions** on control flow based on the state of the particles and the environment
a	Performs **actions** on particles based on state of particles and environment
Information Flow	
□- - - - ·	Reads information from container into control node.
□- - -▶	Pull moves information out of a container into the control nodes local storage during it's operation.
□◀- - -	Writes information from control node into a container.
Control Flow	
⟶	Arrow between **Sampling**, **observer** and **action** nodes to indicate control flow in system.

Table 4: Legend of types nodes and edges used in MetaChem graphs

```python
class CounterDecision(node.Decision):

    def __init__(self, options=2, readcontainers=None, threshold=1):
        if isinstance(readcontainers, list):
            raise TypeError("CounterDecision takes only a single readcontainer")
        if options == 2:
            super(CounterDecision, self).__init__(options, readcontainers)
            self.threshold = threshold
            self.check = 0
        else:
            raise ValueError("CounterDecision takes exactly two control options")
        pass

    def read(self):
        self.check = self.readcontainers.read()

    def process(self):
        if self.check >= self.threshold:
            return self.options[1]
        else:
            return self.options[0]
```

Listing 5: CounterDecision node code

```
Oreaction = control.ClockObserver(Vreactions, Vreactions)
Dupdate = control.CounterDecision(2, Vreactions, 200)
```

Listing 6: Use of clock and counter nodes in StringCatChem

Discussion

We have successfully implemented our mathematical MetaChem framework in a reusable generic manner. With the correct set of system-specific nodes this allows us to define our graph structure and run our algorithm.

We don't provide results based on StringCatChem here. This system is very simple and over time the particles form into a very small number of large particles and stay that way. There isn't much to analyse in this. For results from other systems built on the basis of MetaChem see Rainford (2018); Rainford et al. (2018b).

The independence of our containers' implementation from our control node definitions allows us to implement generic nodes, which can be interchanged between different chemistries and experiments.

As an extension of this we can use subgraphs for common control systems. This immediately includes spatial and time systems e.g. the clock and counter combination described above implements a generational time system.

Our container implementation allows us to use simple storage, such as lists or dictionaries, for smaller/ simpler systems. For larger systems implement a database interface that could handle larger numbers of particles, longer runs, and different particle types.

MetaChem can be used to implement any AChem. A reasonable level of Python knowledge is needed to implement specific unique nodes and particles for a particular AChem.

Once the nodes have been implemented the chemistry itself can be instantiated and run without anything more than basic Python.

In our current implementation, available on GitHub, this is done by implementing the module stringcat_nodes.py, containing class specifications for all nodes required by StringCatChem. We then can run an experiment in this AChem using a script, stringcat_graph.py, that generates instances of nodes and a graph, then runs it for a number of transitions.

Further Work

The implementation of core Static Graph MetaChem code described here means we can start porting other systems to run in our framework. An earlier version of MetaChem is used to combine SwarmChem and JA-AChem (Rainford et al., 2018a). Porting further existing AChems would be the beginning of building a comprehensive code base for AChems. It could be expanded to include analysis tool that work across different systems, allowing direct comparisons of a range of AChems.

Next, we will expand the existing code to allow the graph to change at run time, allowing our AChems to be more dynamic: this is analogous to changing the "glassware" as a chemistry experiment runs, based on the results of reactions. This will use graph transformation rules to allow the graph to change and develop in response to the state of the system.

The naming of MetaChem is not just to imply that it is a meta system for handling AChems. It is capable of being an AChem itself. Nodes are atomic particles, edges are links, and an AChem graph is a composite particle in the MetaChem. Such an approach could allow us to evolve and generate AChems as products of a MetaChem.

Acknowledgements

PFR was funded by a University of York, Department of Chemistry Teaching Studentship.

References

Bersini, H. (1999). Design patterns for an Object-Oriented computational chemistry. In *Advances in Artificial Life*, pages 389–398. Springer.

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

Faulconbridge, A. (2011). *RBN-World: Sub-Symbolic Artificial Chemistry for Artificial Life*. PhD thesis, University of York.

Faulkner, P., Krastev, M., Sebald, A., and Stepney, S. (2018). Sub-Symbolic artificial chemistries. In Stepney, S. and Adamatzky, A., editors, *Inspired by Nature*, pages 287–322. Springer International Publishing, Cham.

Hickinbotham, S., Clark, E., Stepney, S., Clarke, T., Nellis, A., Pay, M., and Young, P. (2010). Specification of the stringmol chemical programming language version 0.2. Technical report, Technical Report YCS-2010-458, Univ. of York.

Hutton, T. J. (2007). Evolvable self-reproducing cells in a two-dimensional artificial chemistry. *Artif. Life*, 13(1):11–30.

Ofria, C. and Wilke, C. O. (2004). Avida: a software platform for research in computational evolutionary biology. *Artif. Life*, 10(2):191–229.

Ono, N. and Ikegami, T. (2001). Artificial chemistry: Computational studies on the emergence of Self-Reproducing units. In *Advances in Artificial Life*, Lecture Notes in Computer Science, pages 186–195. Springer Berlin Heidelberg.

Rainford, P. F. (2019). *MetaChemistry: an algebraic approach to artificial chemistries*. PhD thesis, Department of Chemistry, University of York, Heslington, York YO10 5DG.

Rainford, P. F., Sebald, A., and Stepney, S. (2018a). Modular combinations of artificial chemistries. In *ALife 2018, Tokyo, Japan, July 2018*, pages 361–367. MIT Press.

Rainford, P. F., Sebald, A., and Stepney, S. (2018b). Modular combinations of artificial chemistries. *The 2018 Conference on Artificial Life: A Hybrid of the European Conference on Artificial Life (ECAL) and the International Conference on the Synthesis and Simulation of Living Systems (ALIFE)*, pages 361–367.

Rainford, P. S. M. F. (2018). *Algebraic approaches to artificial chemistries*. PhD thesis, University of York.

Sayama, H. (2011). Seeking open-ended evolution in swarm chemistry. In *2011 IEEE Symposium on Artificial Life (ALIFE)*, pages 186–193.