SCL: An Artificial Chemistry in *Swarm*

Barry McMullin
`mcmullin@eeng.dcu.ie`

Santa Fe Institute
Working Paper Number: `97-01-002`

Dublin City University
School of Electronic Engineering
Technical Report Number: `bmcm9702`

Abstract

This report describes the SCL (v0.04) system. This is an implementation of an artificial chemistry, using the Swarm[1] simulation system. This chemistry is qualitatively based on the system first described in Varela, Maturana & Uribe (1974). This involves three distinct chemical species: Substrate, Catalyst and Link, hence *SCL*. It was designed with a view to generating simple phenomena of autopoietic organisation. Varela et al. included a detailed algorithmic account of their original model; however, as documented in (McMullin 1997), there are a number of problems with interpreting and/or re-implementing that algorithm. Arising from this, that original algorithm was essentially set aside in designing SCL; instead, SCL seeks to capture only the general, qualitative, reaction schemes described by Varela et al.

SCL was developed for two separate purposes. Firstly, it provides a platform to critically re-evaluate the phenomena—particularly autopoietic phenomena—that can be realised with this general kind of reaction scheme. That phenomenological investigation will be described in a separate report. The second objective was to gain experience of the Swarm simulation system and evaluate its suitability for this kind of project. This report is concerned solely with this second objective: i.e. with documenting the implementation, and with evaluating the use of Swarm.

Keywords: Autopoiesis, Artificial Life, Artificial Chemistry, Swarm Simulation System.

[1] `http://www.santafe.edu/projects/swarm`

Contents

1 Introduction

The qualitative inspiration for the chemistry implemented by SCL is the model described in Varela et al. (1974). That model, including the provided algorithm, has been reviewed in detail in (McMullin 1997), and will not be presented again here. Instead, I will briefly describe the qualitative chemical reaction scheme, and outline the way SCL implements it. Note that the *point* of this particular reaction scheme is to model the emergence and stability of *autopoietic organisation*; the reaction scheme was expressly designed by Varela et al. with the specific idea that it would be the simplest possible scheme that could exhibit such organisation. However, the present report is *not* concerned with discussing or elaborating the phenomenology actually generated by the SCL model (autopoietic or otherwise)—that is deferred for a subsequent, separate, report.

2 Overview of the SCL Reaction Scheme

SCL involves three distinct chemical species (implemented as separate *classes*):

Substrate:	`S`
Catalyst:	`K`
Link:	`L`

The SCL reaction scheme is implemented in a 2-D space. Subject to certain constraints, particles move around this space in random walks. The particles can undergo various transformations or reactions; where these involve more than one particle, the reacting particles must be adjacent to each other in the space.

The restriction to a 2-D space is important because it means that linear chains or polymers of `L` particles can actually serve as spatial separators or boundaries. It is stipulated that such chains are permeable to `S` particles, but impermeable to all others. In this way a *closed* chain of `L` particles can "trap" `K` and `L` particles inside, while still allowing `S` particles to move in (and out). These constraints on particle motion are critical to the intended autopoietic phenomena.

SCL supports six distinct reactions, as follows:

1. *Production*:

$$\mathtt{K} + 2\mathtt{S} \rightarrow \mathtt{K} + \mathtt{L}$$

In essence, this is a reaction in which two particles of `S` combine to produce one particle of `L`. However, the reaction can occur only with the mediation of catalyst, `K`. The catalyst particle itself is unaffected by the reaction (hence its name).

2. *Disintegration*:

$$\mathtt{L} \rightarrow 2\mathtt{S}$$

This is a reverse reaction to production. It occurs spontaneously.

3. *Bonding*: Adjacent `L` particles can bond or polymerise into indefinitely long chains. Each `L` particle can form at most two bonds, so that no structures more complex than chains can arise. This is a spontaneous reaction, but it is influenced in a complex way by the precise configuration of `L` particles involved, and by a number of distinct parameters.

`L` chains can break down again either via spontaneous decay of individual bonds (see below), or if constituent `L` particles spontaneously disintegrate back to substrate (when any associated bonds will also undergo forced decay).

4. *Bond Decay*: This is a reverse reaction to bonding, in which individual bonds can decay, breaking a chain. It occurs spontaneously.

5. *Absorption*:

$$\mathtt{L} + \mathtt{S} \rightarrow \mathtt{L}^+$$

In this reaction, an `L` particle can spontaneously *absorb* an `S` particle. The absorbed `S` particle can subsequently be *emitted* again (see below). An `L` which has absorbed an `S` particle is denoted by $\mathtt{L}^+$. An `L` particle can absorb at most one `S` particle—i.e. an $\mathtt{L}^+$ particle cannot undergo a further absorption reaction. The absorption and emission reactions together provide for the permeability of `L` chains to `S` particles.

6. *Emission*:

$$\mathtt{L}^+ \rightarrow \mathtt{L} + \mathtt{S}$$

This is a reverse reaction to absorption, in which a previously absorbed `S` particle is spontaneously re-emitted. An $\mathtt{L}^+$ particle which undergoes disintegration must necessarily first undergo a (forced) emission reaction.

Each of these reactions is controlled by a one or more, user configurable, rate parameters. Similarly, rates of random motion in the space are controlled by separate mobility parameters for each particle class. This parameters are all implemented as class variables of the relevant agent classes.

Of the six reactions, only the first three were present in the original model described by Varela et al. Furthermore, only disintegration was explicitly controlled by a rate parameter, and no explicit mobility parameters were defined, though there were implicit differences in mobility between the different particle classes. The *implementation* of bonding in SCL is quite different from the original.

Bond decay was added to SCL in the interests of making the reaction scheme somewhat more symmetrical. However, if desired, it can be "switched off" simply by setting its rate parameter to zero.

The original model implemented permeability of `L` chains as a special case of *motion* in the space, rather than using the explicit absorption and emission reactions introduced in SCL. However, that suffered from the drawbacks of requiring a neighborhood (for motion at least) extending two cells from any given cell, and also making it more difficult to control the rate at which permeation could occur *separately* from general mobility rates. I suggest that introducing the explicit absorption and emission reactions makes the model easier to understand and provides more intuitively tractable parameters.

3 Overview of the SCL Implementation

SCL is implemented using the Swarm Simulation System[2]. That is to say, SCL is written in Objective-C, using the libraries supplied by Swarm, and is intended to be portable across the computing platforms supported by Swarm (Unix, running X-window and the `gcc` compiler).

In what follows, it is assumed that the reader has a basic understanding both of Objective-C and Swarm.

Unless explicitly stated otherwise, all references to "random" selection here imply selection with a *uniform* probability function over the relevant set.

3.1 Space and Time

SCL implements the reaction scheme in a discrete, 2D, space with toroidal topology. Time is also discrete. The space is organised as a square lattice. The dynamics occur based on either the Moore (9-cell) or the von Neumann (5-cell) neighborhood (this is, to some extent, configurable). Each cell in the space is occupied by a particle object. There are 4 subclasses of particle objects, allowing for each of the three distinct chemical species and a "hole" particle class (denoted `H`). Bonds are also represented by objects of another separate class. Lists are maintained of all particles of each class, and of all bonds. The basic execution cycle of the model, for one timestep, is that each such list is traversed in turn, and each particle or bond is sent a `step` message. The effect of this message is, of course, class specific, but generally involves possible motion (by swapping position with an adjacent particle) followed by one or more possible reactions.

In the following more detailed descriptions, the particle which is actually executing its `step` method will be called the *primary* particle, to distinguish it from the neighboring, or *secondary*, particle(s) which it may interact with.

3.2 Motion

Motion for all particle classes follows the same general strategy. In essence, particles undergo random walks in the space. The relative rates of movement for the different particle classes are controlled by user configurable parameters.

Note that, on any given timestep, each particle gets exactly one opportunity to be the primary particle of a movement; but it will also get several (actually four) opportunities to act as the secondary particle of a movement.

In more detail, movement takes place as follows:

1. Firstly, particles may, at any given time, be either mobile or immobile. The exact circumstances governing this are class specific. However, for all classes, once a particle moves, it then becomes immobile for the remainder of that timestep. That is, no particle may move more than once within a single timestep.

2. Assuming a given primary particle *is* mobile, a neighboring cell is chosen at random from the von Neumann neighborhood. The particle in this cell is then the *secondary* particle—a candidate for swapping with the target particle.[3] The mobility status of this candidate particle is checked.

[2] `http://www.santafe.edu/projects/swarm`

[3] It follows from this choice of neighborhood that each particle gets exactly four opportunities to be considered as the secondary particle of a movement, per timestep. The von Neumann neighborhood is used here because this makes implementation of *im*-permeability of the `L` particle chains relatively simple.

3. Assuming that the neighboring particle *is* also mobile, a bernoulli random variable (biased coin toss) is evaluated to determine whether to proceed with the swap or not. The probability parameter used in this evaluation thus allows control of the relative rates of movement.

   In an earlier implementation of SCL, this probability of movement was simply a parameter of the class of the primary particle. However, this is not completely satisfactory because a particle may move as a result of being a primary particle or as a secondary particle—and, on that original scheme, the probabilities in the two cases, for the same two particles, could be different, which is somewhat counter-intuitive.

   In SCL v0.04 each particle class still has a class specific mobility parameter, called the `MobilityFactor`. But rather than simply taking the probability to be the `MobilityFactor` of the primary particle class, it is calculated as the geometric mean of the `MobilityFactor` parameters for the classes of the primary and secondary particles. This means that the probability is always the same for swapping particles of any two particular classes, regardless of which is primary and which is secondary. Further, by using the geometric (rather than, say, arithmetic) mean, we ensure that if the `MobilityFactor` is set to *zero* for either class, then the overall probability is guaranteed to be zero. Finally, of course, if the `MobilityFactor` for both classes is the same, then this just *is* the probability of the particular swap.

   Note that the primary reason for introducing the `H` particle class was so that this movement strategy could be applied uniformly. The alternative would be that holes in the space would be represented by positions with no object present at all (i.e. the Objective-C `nil` object pointer)—but then there would be no corresponding `MobilityFactor` and nothing to "swap" with either. While using explicit `H` particles instead presumably involves some performance penalty, I considered this worthwhile in order to keep the implementation simpler and more consistent.

4. Assuming that the bernoulli random variable evaluates as `YES`, the two particles swap position. Both then become immobile for the remainder of that timestep.

`L` particles with one or more bonds are always immobile. This is a very strong restriction on the model. It was present in the original model of Varela et al. and is retained here. The presumed reason for this restriction is to avoid the rather complicated issue of what should happen to a bond (and the `L` particle at its other end) if a bonded `L` particle is allowed to move.

3.3 Reactions

The reactions are all initiated and directed by the execution of the `step` method on some particular primary particle or on a bond. Which reactions are considered is determined by the particle class. The production reaction can be initiated only by `K` particles; the bond decay reaction is initiated by the bond itself; all other reactions are initiated only by `L` particles.

Reactions typically occur in either the von Neumann or Moore neighborhood of a given primary particle. A user configurable parameter determines a single *default neighborhood* which applies to all such reactions.

3.3.1 Production

Production is initiated by a primary `K` particle, and is implemented as follows:

1. A bernoulli random variable is evaluated with probability parameter `productionProbability` (the rate parameter for the reaction). If this evaluates as `NO`, the reaction terminates.

2. A random neighbor is chosen from the default neighborhood (`neighborParticle0`). If this is *not* an `S` particle, the reaction is terminated.

3. There will be exactly two cells in the default neighborhood which are immediately adjacent to both the primary `K` particle and the `neighborParticle0` `S` particle. One of these is chosen at random (`neigborParticle1`). If this does *not* contain an `S` particle, the reaction is terminated.

4. `neighborParticle0` is replaced with an `H`; `neighborParticle1` is replaced with an `L`. That completes the reaction.

3.3.2 Disintegration

Disintegration is implemented by a primary `L` particle, and happens in two quite separate stages. Each `L` particle has an associated boolean state variable called `disintegrating`. When an `L` particle is initially created (via production) this is set to NO. The first phase of disintegration is for this state variable to change to YES, and happens as follows:

1. A bernoulli random variable is evaluated with probability parameter `disintegrationProbability` (the rate parameter for the reaction). If this evaluates as NO, the reaction terminates.

2. The `disintegrating` state variable is set to YES. This completes the first stage of the reaction.

Once an L particle goes into the disintegrating state it stays in that state until the disintegration can be completed. In general it cannot be completed immediately because it relies on finding one or more adjacent holes to accommodate the disintegration products.

An L particle in the disintegrating state attempts to complete the reaction as follows (this attempt occurs for the first time on the *same* timestep as the L particle goes into the disintegrating state; it will be repeated, as necessary, on each following timestep, until the reaction is completed):

1. If the L particle has an absorbed S particle (i.e. it is strictly an L^+), then an attempt is made to forcibly emit this S. The details of this process are given in section 3.3.6.

2. If the L particle still has an absorbed S particle (i.e. the attempted emission failed) then the reaction is terminated.

3. A random neighbor is chosen from the default neighborhood (`neighborParticle`). If this is *not* a H, the reaction is terminated.

4. The H is replaced with an S particle.

5. Any bonds associated with the primary L particle are forced to decay (see section 3.3.4).

6. The primary L particle is replaced with an S particle. This completes the disintegration reaction.

Note finally that once an L particle has gone into the disintegrating state it will no longer undergo the absorption reaction (since this could further delay the completion of the disintegration); however, it can still participate in bonding.

3.3.3 Bonding

Bonding is initiated by a primary L particle; however, in general, the various constraints or influences affecting bonding are evaluated symmetrically for both the primary and secondary L particle.

The factors influencing the bonding reaction are as follows:

- Bonding is inhibited if an L particle has already formed two bonds. This sets the maximum number of bonds to any L particle as two, and ensures that the only particle aggregates or "molecules" which can form are linear "chains" rather than any more complex, branching, structures.

- Two other mechanisms or interactions by which bonding can be inhibited are also provided. Each of these is qualified by a BOOL parameter whereby they can be enabled or disabled. These are essentially intended as alternative mechanisms for preventing "premature" bonding of free L particles that have been produced within the boundary of an autopoietic entity. However, even though the typical mode of operation would be with just one or the other of these interactions enabled, SCL will actually allow both to be simultaneously disabled, or both simultaneously enabled, if desired.

  The need for *some* such form of inhibition interaction is quite subtle but, it transpires, quite crucial to the autopoietic phenomenology of the model; while no such effect was mentioned explicitly in (Varela et al. 1974), it is almost certain that it was present in their model. This is discussed further in (McMullin 1997).

  - *Inhibition by a neighboring L Chain*: This interaction allows bonding to be inhibited to any L particle which is *alongside* (as opposed to, at the end of) an existing chain of L particles. This is implemented for any given L particle by checking whether there are any existing doubly bonded L particles in its (Moore) neighborhood. This interaction is qualified by the BOOL parameter, `chainInhibitBondFlag`. If this is YES this inhibition effect is enabled; if it is NO then it is disabled.

    As discussed in (McMullin 1997), this particular form of bond inhibition reaction was inspired by examination of the FORTRAN IV code for one early version of the model developed by Varela et al.

  - *Inhibition by a neighboring* `K` *particle*: This is implemented for any given `L` particle by checking whether there are any `K` particles in its (Moore) neighborhood. This interaction is qualified by the `BOOL` parameter, `catInhibitBondFlag`. If this is `YES` this inhibition effect is enabled; if it is `NO` then it is disabled.

    This alternate form of bond inhibition reaction was inspired by (Lizana 1981).

- A second bond to any particular `L` particle must form an angle of 90° or more relative to the first bond. This rules out forming two bonds "on top of each other" (a double bond between the same two `L` particles), or with an angle of just 45° between them.

- Given that a new bond would satisfy the above constraints for both affected `L` particles, the formation of the bond is controlled by a reaction rate, or probability parameter. More precisely, one of three possible rate parameters is applied, depending on the exact conformation of the proposed new bond:

  1. If neither `L` particle already has a bond, the new bond would initiate a new chain; in that case the probability is set by the parameter `chainInitiateProbability`.
  2. If just one of the two `L` particles already has a bond, then the new bond would extend an already existing chain by one more particle; in that case the probability is set by the parameter `chainExtendProbability`.
  3. If both of the `L` particles already have a bond, then the new bond would splice together two existing chains; in that case the probability is set by the parameter `chainSpliceProbability`.

The detailed implementation of the bonding reaction is as follows:

1. If bonding to the primary `L` is inhibited due to already having two bonds, or either of the inhibition interactions (an adjacent doubly bonded `L` particle, qualified by `chainInhibitBondFlag`, or an adjacent `K` particle, qualified by `catInhibitBondFlag`) the reaction is terminated.
2. A random neighbor (`neighborParticle`) is chosen from the default neighborhood.
3. If the primary `L` already has a bond to this neighbor, or to either of the neighbors at an angle of 45° to this neighbor, the reaction is terminated.
4. If `neighborParticle` is not an `L` particle the reaction is terminated. Otherwise it becomes the secondary `L` particle in the reaction.
5. If bonding to the secondary `L` is inhibited due to already having two bonds, or either of the inhibition interactions, the reaction is terminated.
6. If the secondary `L` already has a bond to either of its neighbors at an angle of 45° to the direction from it to the primary `L` particle, the reaction is terminated.
7. If neither the primary nor secondary `L` particles already have a bond, `bondingProbability` will be set to `chainInitiateProbability`; if just one of them already has a bond, `bondingProbability` will be set to `chainExtendProbability`; if both of them already have a bond, `bondingProbability` will be set to `chainSpliceProbability`.
8. A bernoulli random variable is evaluated with probability parameter `bondingProbability`; if this evaluates as `NO`, the reaction terminates.
9. The secondary `L` particle creates the bond to the primary `L` particle (and passes a pointer to it back to the latter).
10. The primary `L` particle records the presence of the new bond also. This completes the bonding reaction.

3.3.4 Bond Decay

Bond decay can happen either as a spontaneous reaction, or as a forced reaction when an `L` is completing the disintegration reaction. The spontaneous reaction is initiated by a bond, and is implemented as follows:

1. A bernoulli random variable is evaluated with probability parameter `bondDecayProbability` (the rate parameter for the reaction). If this evaluates as `NO`, the reaction terminates.
2. The bond decays (i.e. the two affected `L` particles are told to remove the bond). This completes the spontaneous bond decay reaction.

The forced bond decay reaction is initiated by one of the bonded `L` particles, as it completes the disintegration reaction. It is implemented as follows:

1. The primary `L` particle tells the bond to decay.
2. The bond decays (i.e. the two affected `L` particles are told to remove the bond). This completes the forced bond decay reaction.

3.3.5 Absorption

Absorption is initiated by a primary `L` particle, and is implemented as follows:

1. If the primary `L` particle already has an absorbed link (i.e. it is strictly an $\mathtt{L}^{+}$ particle) then the reaction terminates.
2. A bernoulli random variable is evaluated with probability parameter `absorptionProbability` (the rate parameter for the reaction). If this evaluates as `NO`, the reaction terminates.
3. A random neighbor is chosen from the default neighborhood (`neighborParticle`). If this is *not* an `S` particle, the reaction terminates.
4. `neighborParticle` is replaced with a hole.
5. The primary `L` particle is marked as having an absorbed substrate—i.e. it changes to the $\mathtt{L}^{+}$ form. That completes the absorption reaction.

3.3.6 Emission

Emission is initiated by a primary `L` particle. It can happen either as a spontaneous reaction, or as a forced reaction when an `L` is in the disintegrating state. The spontaneous reaction is implemented as follows:

1. If the primary `L` particle does not have an absorbed link (i.e. it is not in the $\mathtt{L}^{+}$ form) then the reaction terminates.
2. A bernoulli random variable is evaluated with probability parameter `emissionProbability` (the rate parameter for the reaction). If this evaluates as `NO`, the reaction terminates.
3. A random neighbor is chosen from the default neighborhood (`neighborParticle`). If this is *not* a hole (`H`), the reaction terminates.
4. `neighborParticle` is replaced with an `S` particle.
5. The primary $\mathtt{L}^{+}$ particle is marked as no longer having an absorbed substrate—i.e. it reverts to the `L` form. That completes the absorption reaction.

In the case where the emission reaction is forced, as a side effect of the disintegration reaction, the implementation is the same as this spontaneous reaction except that steps 1 and 2 above are bypassed.

4 Getting SCL

SCL is available only as a source archive. See section 7 for information on retrieving it. Unpack the archive file into an empty directory. Edit the `SWARMHOME` variable in `Makefile` as appropriate, and type `make`. Provided Swarm itself has been correctly installed on your system (and is a compatible release) then SCL should now be built.

SCL v0.04 was originally developed against the Swarm release dated `961002`; since this is still a *beta* release, later releases will not necessarily be backward compatible, and SCL may well require modification to successfully build against any such later Swarm release.

5 Using SCL

When SCL starts up it initially creates three windows to facilitate user interaction:

- `Swarm`
- `ParameterManager`
- `WorldManager`

The latter two of these are Swarm `probeDisplay` windows on specifically designed user interaction objects; the standard Swarm documentation provides general information on the facilities and interactions offered by `probeDisplay` windows, and that will not be repeated here. These initial three windows are required throughout a session with SCL, and none should be deleted by the user.

Once a model world is created or loaded from disk, two additional windows are created by SCL:

- `SCL World`
- `UserCellEditor`

All of these user interaction windows are discussed in detail in the following sections.

5.1 SWARM Window

This is the standard Swarm *control panel* window. It provides buttons to start the simulation (`Go`), stop it (`Stop`), single step (`Timestep`) and terminate the simulation (`Quit`). The buttons for controlling execution of the simulation will, of course, operate only while a simulation model, or world, exists. The tools for creating and manipulating these world models will be described separately below. This window is *not* generated as a `probeDisplay` on some object, but is rather generated automatically as a standard feature of Swarm.

5.2 ParameterManager Window

This is a (tailored) `probeDisplay` on the `parameterManager` object. It provides facilities to monitor and modify all the model parameters supported by SCL, and for saving and/or reloading these parameters to or from disk files.

To modify a parameter, simply click on the relevant position in the window and edit the value. Ensure that you press `Enter` or `Return` to cause the new value to take effect (this is a standard feature of Swarm `probeDisplay` windows). Note that, because the `probeDisplay` is continuously refreshed while the simulation is running, it is only practical to alter parameters while the simulation is stopped.

Note also that no attempt has been made to make the file interface "robust"; specifically, any error in accessing a file (e.g. if it does not exist and/or cannot be created, or is of the wrong format etc.) will result in a fatal exception being raised and SCL will exit.

5.3 WorldManager Window

This is a (tailored) `probeDisplay` on the `worldManager` object. It provides "high level" facilities for creating and manipulating a model world (or, more precisely, a model world *state*), and for saving and/or reloading the world state to or from disk files.

Note that if the world state is loaded from disk then any already loaded world model will be automatically (and silently) dropped first. As with the file interface for parameters, no attempt has been made to make the world state file interface "robust", and any error in accessing a file will result in a fatal exception being raised and SCL will exit.

`PMMLCG` refers to the pseudo random number generator object used to implement the (pseudo!) stochastic effects in SCL. The `WorldManager` window allows the exact state of this generator to be monitored and/or modified. This permits, for example, a given world state to be reloaded and then restarted with a different initial value for the generator state, thus potentially exploring the robustness of stochastic phenomena in the model.

The `worldTime` records the number of full timesteps which have elapsed. It is meaningful only when a world model has been created or loaded from disk.

The `WorldManager` window offers two ways of creating a new model world. As with loading a model from disk, if a new model is created here then any already loaded model will be automatically (and silently) dropped first. The `createDefaultWorld` method provides a "canned" world with certain compiled-in characteristics. The `createEmptyWorldWithX:Y:` method allows a world of arbitrary size to be created. This will be initially "empty", i.e. populated exclusively with holes.

Once a world has been created (or loaded from disk) particles can be incrementally added to (or removed from) the world using the three `adjust...NumberBy:` methods—these accept a positive or negative integer argument, being the number of particles to add or remove respectively. Particles are added or removed at random positions with these methods. They can be added only into holes; when they are removed, they leave holes. These methods return the number of particles *actually* added or removed. This may generally differ from the number requested. The methods can only add new particles into existing holes—so if the world becomes full, no more can be added. Conversely, only existing particles can be removed—so once all particles of a given class have been removed no more can be taken out.

Special methods are also provided to forcibly fill the space completely with substrate, or completely empty it again (i.e. "fill" with holes).

5.4 SCL World Window

This is a graphic image of the state of the model. This is implemented with the Swarm `ZoomRaster` class, and so can be magnified or reduced (by dragging any corner). Note however that this does *not* allow "windowing" onto a subarea of the world model, or panning across it: this window will always display the entire world. If this window is resized while the model is stopped the image will *not* be automatically updated; you can use the `update` button in the `WorldManager` probe to force an update.

`S` particles are shown as small, outlined, red squares; `K` particles are shown as larger, filled, yellow squares. Bonds are blue line segments. `L` particles are normally shown as blue, outlined, squares, the same size as `K` particles. If an `L` particle goes into the disintegrating state then it turns grey. If an `L` particle has an absorbed substrate (i.e. it is in the L^+ form) then this absorbed particle is still shown as a small, outlined, red square, but internal to the `L` particle.

Moving the pointer over this window and clicking mouse button 1 (normally the left button) will set the co-ordinates of the `UserCellEditor`—which is discussed in section 5.5 below.

Moving the pointer over this window and clicking mouse button 3 (normally the right button) will create a new `probeDisplay` for the particle object pointed at. These particle objects will not be detailed further here—consult the source code to see exactly what instance variables and methods are associated with any particular particle class. However, it is worth noting that this is the *only* mechanism for a user to manipulate or modify *bonds*—i.e. by creating a `probeDisplay` on an affected `L` particle, and using the relevant methods provided there. There is *no* mechanism to create a `probeDisplay` for a bond object.

Be aware that, due to an underlying problem with the current release of Swarm, SCL will crash if any object, which is being probed with a `probeDisplay`, is dropped. Since particle objects in Swarm may quite generally get dropped, it is recommended that you only a create `probeDisplay` on a particle object while the simulation is stopped; and that you drop this `probeDisplay` (using the special, red, destroy button—*not* your window manager) before continuing execution of the model.

5.5 `UserCellEditor` Window

This is a (tailored) `probeDisplay` on the `userCellEditor` object. It provides "low level" facilities for manipulating the model world. In particular it allows the user to "force" a particle of any class at any cell in the world. It is used by first shifting its coordinates to the desired cell—by clicking mouse button 1 while pointing at the cell, as mentioned above—and then forcing the desired particle class at that cell. If you wish to modify the particle in any further way (especially to create bonds between `L` particles) you must create a `probeDisplay` on the particle itself and use that, as already discussed.

6 SCL Design Notes

SCL demonstrates possible approaches to some Swarm programming issues that are not covered in the sample applications distributed with the current (`961002`) release of Swarm, and which may therefore be of interest to the broader Swarm community. The definitive reference here is, of course, to the source code itself. But a brief introduction will be given in the following sections.

6.1 Who needs an Observer?

The standard Swarm example applications enforce a strict distinction between an "observer" and the "model" itself. The idea is to facilitate easier switching between versions which operate in "batch" mode (with no user interaction) and versions which operate in "GUI" mode (with user interaction).

This is a very reasonable and useful strategy.

However, in the case of SCL I knew that I did not want the facility to operate in "batch" mode—it was intended and designed strictly as an interactive application. Furthermore, I found that the attempt to maintain the flexibility of supporting both batch and interactive modes made the application more complex, and also made the implementation of user interaction more cumbersome. Therefore I abandoned the observer/model distinction, and rationalised the program accordingly. This is not an intrinsically better or worse approach; it is simply another alternative that some users may find of interest.

6.2 Class Variables

Objective C does not strictly offer "class variables", or, at least, has no explicit support for them. However the effect of such variables *can* be realised by using static variables with file scope in the class implementation file. Note, however, that these variables will *not* be accessible to subclasses.

Class variables are useful in SCL to implement reaction and mobility parameters. These must be common to all objects of a specified class. They could be implemented with conventional global variables; but since they *are* class specific, it is more appropriate to implement them as class variables.

A problem with this approach is that the Swarm probe machinery does not support probing of class objects. Thus, probes cannot be used *directly* to give user access to these parameters. I understand that this is a fairly fundamental limitation, and is unlikely to be removed in the future.

However, this is easily worked around by introducing a special object, `parameterManager` which "shadows" the class variables as instance variables. This can then be probed as usual to provide access to the parameters. Some care is required in using this approach. One can normally use a probe to directly modify instance variables; one can do that in this case also, but it will not automatically have the desired or expected effect, because such modifications do not, in themselves, affect the class variables. To correct this, the `parameterManager` provides the `apply` method which essentially copies all the instance variables into the corresponding class variables. To make the behaviour as robust as possible, an invocation of `apply` in included in *every* timestep—so that any changes in the instance variables will automatically take full effect at the start of the next timestep. Similarly, `apply` is invoked immediately prior to saving a model to disk.

6.3 2D Space/2D Agents

Swarm provides a standard class (`Discrete2d`) for implementing a discrete, 2-D, space, with an arbitrary object (or `nil`) in each cell. This class does not provide for any special treatment or handling of the boundary of this space. A subclass of this, `Grid2d` adds functionality to check that a new object is not inserted in the space overwriting a previous object. The sample `heatbugs` application uses this class, and additionally imposes a toroidal topology. The calculations for the latter are carried out in the `heatbug` agents; because these calculations are rather frequent, the size of the space is cached in the agents.

In SCL I wanted both this toroidal topology, and also the restriction to at most one agent (particle in my case) in any single cell. However, the approach in SCL is somewhat different—indeed, it takes an almost diametrically opposed approach to where to locate these functionalities.

First, instead of locating the calculations for the toroidal topology in the agents, I developed a new space class, `DiscreteToroid`, subclassed from `Discrete2d`, which offers explicit methods for adjusting (wrapping) co-ordinates appropriately. I consider this preferable, because the space object is the appropriate object to "know" how to deal with co-ordinates that go outside the defined boundaries. However, it does mean sacrificing some performance compared to the `heatbugs` approach. In my particular case, performance has not been a problem, so this was a reasonable change; it would not necessarily be so in other models.

Secondly, instead of having the space attempt to restrict occupation of any cell to one object at a time, I embedded this functionality in a new agent class, `Agent2d`, which became the superclass for all particle classes. `Agent2d` also encapsulates a lot of other "generic" functionality for agents in a 2D space—at least where interactions are local (in the von Neumann or Moore neighborhoods), and include movement by swapping. `Agent2d` may thus be useful in other models, quite different from SCL. `Agent2d` functionality is layered, in part, on top of two other new agent classes, `VNneighbor` and `MooreNeighbor` which provide support for co-ordinate calculations within these particular neighborhoods.

6.4 Mortal Agents!

It is in the nature of the SCL model that agents (particles) are created and destroyed on an on-going basis during the simulation. This led to two separate problems.

Firstly, there is a general problem with Swarm that the application will crash if any object which is being probed (with a `probeDisplay`) is dropped. I have not attempted to provide any fix for this. It can be worked around, as already explained, by taking care to ensure that `probeDisplay` windows on particles are created only while the simulation is stopped, and are dropped before the simulation is continued. It would probably be possible to *enforce* this restriction from within the program; but, since this will probably be fixed "properly" in a future Swarm release, I did not consider that the effort was justified.

Secondly—and more seriously—the scheduling of model execution can be disrupted, leading to a crash, if agents are dropped during the simulation. This arises where a list of agents is being traversed (via a `ForEach:` method), and, within that traversal, an object is removed from the list. This is a consequence of the way lists are currently implemented in Swarm. Again, the restriction should be removed in a future Swarm release. However, since this behaviour was essential to SCL it was absolutely necessary to develop a workaround.

The strategy adopted is to have a "manager" object for the agents. This accepts requests to create or drop agents and effectively puts these requests into a *queue*. A certain amount of processing of the request can be (indeed, must be) done immediately; but the list of agents which is potentially being traversed by `ForEach:` is not immediately updated. Instead, that final update is deferred until some time later, when it is known to be safe.

This mechanism is implemented in SCL via a generic `AgentManager` class; this is separately instantiated to handle each of the different classes of agent required (one for bonds, and one each for the different particle classes, including holes).

This mechanism works, but is quite cumbersome and somewhat error prone (care is required in ensuring that the various queues are processed when necessary but also when safe). It should probably be dispensed with as soon as Swarm can handle agent dropping safely. On the other hand, this mechanism does offer an interesting potential (*not* currently exploited) for improving performance. The `AgentManager` could quite easily arrange to *cache* dropped agents, and recycle them in response to new creation requests. This might significantly reduce the overhead associated with creating and/or dropping agents. However, some care would be required to ensure that this "pseudo" creation was equivalent to "genuine" creation. This latter issue is potentially complicated by Swarm's powerful, but complex, "create phase protocol".

6.5 Agent Update Order

SCL implements a single timestep by processing each agent list in turn, issuing a `step` message to each agent on that list. The order in which the different agent lists (or classes) are processed is fixed. *Within* each agent list, processing is done in the list's "natural" order. For those agent classes where agents get dynamically created and dropped (i.e. all classes other than `K`), the ordering of the corresponding agent list will thus suffer some degree of automatic permutation or shuffling as the simulation proceeds; however, this will be fairly slow.

As a general rule, in this kind of simulation, it would be preferable to randomise the order in which agents are updated on every timestep. This would give a greater assurance that accidental artifacts will not arise due to a particular update order. As can be seen, SCL v0.04 falls somewhat short of this ideal. However, given that there is *some* randomising of the update order (albeit relatively slow and unsystematic), and, more importantly, that the update order is not systematically correlated with the spatial positions of the agents, it is probably satisfactory. It could be improved by explicitly executing a random permutation on each agent list on each timestep. There is no built-in support for this in the current Swarm libraries; however, it would not be all *that* difficult to implement, and should have only a moderate effect on performance of the model.

With the wisdom of hindsight (!), SCL could arguably be implemented perfectly satisfactorily on the basis of randomly sampling agents for update (similar to a monte carlo technique), rather than having a guaranteed update of every agent on every ("macro") timestep. This would eliminate any artifacts of update order. As an added bonus, since there would no longer be a `ForEach:` method scanning an agent list, some of the agent management problems might also be bypassed.

6.6 Persistent Storage

An essential part of the requirement for SCL was to support persistent storage of a model—i.e. being able to save a model into a disk file, and reload it later. This is not supported by any of the current Swarm example applications, and there is little support for it built into Swarm.

Swarm does provide object classes `ObjectSaver` and `ObjectLoader` intended to take an arbitrary object and save it or reload it. I experimented with these, but eventually found them too cumbersome. They operate by exploiting the Swarm probe machinery. Thus, in order to control exactly what instance variables are processed, it is necessary to establish appropriate `probeMaps`. Further it is difficult to use these methods in a way which easily respects the class inheritance hierarchy. Finally, in some cases, instance variables cannot be meaningfully saved to disk in their normal form—because they are actually pointers, and the same object will not generally be guaranteed to be at the same location when the model is reloaded (indeed, it would generally be highly unlikely). In these cases it is necessary to "re-code" these variables in some sense. To do this within the `ObjectSaver` and `ObjectLoader` framework would mean developing special "proxy" objects, to which `ObjectSaver` and `ObjectLoader` could be applied, but which would contain the "recoded" instance variables. Given all these complications, I abandoned this approach.

The strategy I adopted was to provide all relevant objects with methods, `saveTo:` and `loadFrom:`, which will save to or load from a given disk file (implemented as a Swarm `OutFile` or `InFile` object as appropriate). The `sclModelSwarm` then provides methods to get a file name, open the file, and invoke the required `saveTo:` and `loadFrom:` methods, in order, on all required objects. This finally worked satisfactorily, but there were various problems along the way:

- The `InFile` and `OutFile` classes did not include methods for all native types—specifically, `unsigned` and `BOOL`. These were saved in `int` form instead, but this is not robust and obscures the code.

- The saving and loading methods are not symmetrical; it is necessary to write extra white space to guarantee correct reading.
- `InFile` and `OutFile` do not record the *types* of the fields being stored, which makes it harder to detect mismatches between the saving and loading processes.
- The `WorldRaster` class has a bug, such that, when dropped, it does not automatically cause the corresponding widget to be dropped.
- To guarantee following exactly the same trajectory after reloading a model, it is necessary to store (and reload) the state of the underlying random number generator. This is not directly possibly with the standard Swarm generator. This is discussed in more detail in the next section.
- Similarly, to guarantee following the same trajectory on reloading, it is necessary that the ordering of the agent lists be preserved. This is somewhat tricky. It means, in particular, that the SCL model must *not* be saved simply by scanning the 2D space, and saving the particles in that order; rather it must be done by scanning the agent lists, and saving in *that* order instead.
- There is still an issue of how, exactly, to "recode" pointer information. In SCL the only significant case of this was the pointers from bond agents to `L` agents and *vice versa*. The solution in this case was simply to recode the pointers to `L` agents by their world co-ordinates. The pointers could then be reconstructed on reload (provided, of course, that the `L` particles are reloaded, without any bond information, *prior* to the bonds). The reverse pointers, from `L` particles to bonds, can also then be re-constructed as the forward pointers are reestablished.
- There is a significant issue of what, if any, state information is hidden in the Swarm scheduling and activity machinery. There is no built-in support for accessing this. In the case of SCL, which does not make use of any advanced features of this machinery, I finessed this problem by having the only activity be to execute the `step` method on `sclModelSwarm`. This object (and method) are not changed by saving and/or reloading; this variation is all hidden in other SCL objects, which the activity machinery is not aware of. One small side effect of this is that the "model time" cannot be taken from the Swarm activity system (as it continues to increase monotonically, regardless of saving and loading operations). Instead, the `worldManager` simply keeps its own independent record of model time, which *is* saved and/or reloaded as necessary.

6.7 Access to Random State?

Swarm provides a random number library, including a variety of underlying generators. The `PMMLCG1` generator is made available (via the globally visible `uniformRandom` object) by default. A problem with this arrangement is that the underlying `PMMLCG` object does not provide access to its internal state. This makes it impossible to stop a simulation and save it to disk in such a way that, when restarted, it will continue along exactly the same, deterministic, trajectory. One may query whether this is necessary or not: in a stochastic simulation such as SCL, any phenomena of real interest ought to be robust and statistically repeatable anyway. Nonetheless, for debugging and demonstration purposes, it is often convenient to be able to reproduce a model trajectory exactly, and I considered that an essential feature in SCL. I worked around this deficiency simply by creating a new object class, `SCLRandomBit`, subclassed from `PMMLCG1`, which provided methods to save and load the exact state of the generator to or from a disk file. Methods are also provided to access and/or set the state, for good measure; in SCL these are used to allow the `worldManager` (and thus the user) to monitor or alter the state if desired.

6.8 Agent Graphics

The SCL model has five distinct classes of agents (bonds, holes, `S`, `K`, `L`) which must be displayed on a 2D raster. Bonds and `L` particles have further variations in how they should be displayed (four bond orientations, stable versus disintegrating `L` particles, `L` versus $\mathtt{L}^+$ particles).

In SCL v0.04 these display issues have been addressed in extremely crude ways. Each agent (and variation) is represented by some form of bitmap, but these bitmaps are generated, on the fly, pixel-by-pixel, by dedicated, hard wired, code. This is the end point of an incremental design process, in which the display facilities were gradually evolved and made more complex. The final result is functional; but it is also extremely cumbersome and fragile (i.e. it is difficult to modify without causing unintentional side-effects). Performance is probably also very poor.

However: in the current release of Swarm, the only documented methods for drawing on rasters are for coloring individual pixels, or rectangles. This is a significant limitation. I would hope that future releases will provide more sophisticated 2D graphics primitives—in particular, making it easy to display arbitrary bitmaps. In the meantime, SCL v0.04 graphic handling is *not* a good example to emulate.

7 Conclusion

The artificial chemistry implemented by SCL v0.04, and described here in detail, was qualitatively inspired by the system described in (Varela et al. 1974). However, that original *algorithm* was deliberately *not* used in the development of SCL. This arose partly due to the difficulties of interpretation previously documented in (McMullin 1997); partly because I wanted to significantly expand the parameter space of the original model; and partly because, given the object-oriented nature of the Swarm system, it seemed more appropriate to develop a new, properly object oriented, implementation.

This report has *not* offered any review of the phenomenology presented in SCL, or of any particular experiments carried out with it; that is deferred to a subsequent report. This report has been limited rather to documenting the implementation of SCL and, in particular, documenting the use of Swarm for this purpose. I hope it may be of some value to other researchers, possibly in quite different fields, but interested in using the Swarm system.

I have found the Swarm system a good platform for this development. It was especially beneficial in making it relatively easy—via the probe mechanism—to provide good user interface functionality. The object oriented, agent-based, architecture proved particularly suitable for SCL, in that the structure of the code is able to closely mimic the structure of the qualitative description. The Swarm release in use was still only at beta stage, and was clearly deficient in certain respects. Notwithstanding that, the general quality and reliability were high. The documentation is also already quite extensive and usable, though still incomplete.

In conclusion, it seems to me there are already good grounds for believing that when the full, stable, release of Swarm becomes available, it will be a very useful and beneficial tool for scientific simulation of agent based systems.

References

Lizana, R. (1981), Modelo Bidimensional de Reproduccion como Fractura de una Unidad Autopoietica, Master's thesis, Universidad De Chile, Facultad de Ciencias, Departamento de Biologia. In Spanish.

Maturana, H. R. & Varela, F. J. (1980), *Autopoiesis and Cognition: The Realization of the Living*, Vol. 42 of *Boston Studies in the Philosophy of Science*, D. Reidel Publishing Company, Dordrecht, Holland. With a preface to 'Autopoiesis' by Stafford Beer. Series editors: Robert S. Cohen and Marx W. Wartofsky.

McMullin, B. (1997), Computational Autopoiesis: The Original Algorithm, Technical Report `97-01-001`, Santa Fe Institute, Santa Fe, NM 87501, USA. `http://www.santafe.edu/sfi/publications/Working-Papers/97-01-001/`

Varela, F. J., Maturana, H. R. & Uribe, R. (1974), 'Autopoiesis: The Organization of Living Systems, its Characterization and a Model', *BioSystems* **5**, 187–196.

Acknowledgments

This report was written while I was a visiting researcher (1996/97) with the Swarm Project Group at the Santa Fe Institute. This visit was made possible by generous support from Dublin City University and the Swarm Project. Swarm has been supported by Grant No. N00014-94-1-G014 from the Department of the Navy, Naval Research Laboratory, acting in cooperation with the Advanced Research Projects Agency. Swarm has also benefited from earlier support from The Carol O'Donnell Foundation and from Mr. and Mrs. Michael Grantham.

I am indebted to the School of Electronic Engineering in DCU (particularly through the agency of its Head, Charles McCorkell) for continuing encouragement.

All errors remain, of course, my own responsibility.

The html version of the article could not have been created without the benefit of the **LaTeX**2HTML[4] translator developed by Nikos Drakos[5] and the Harvard[6] extensions for author/date citations developed by Peter Williams[7] and Thorsten Schnier.

Revision History

First published in January 1997, simultaneously as Working Paper No. 97-01-002 of the Santa Fe Institute and as Technical Report No. bmcm9702 of the School of Electronic Engineering of Dublin City University.

Author Contact Information

Barry McMullin

Santa Fe Institute
(Visiting Researcher 1996/97)
1399 Hyde Park Road
Santa Fe, NM 87501
USA

Voice: +1-505-984-8800
Fax: +1-505-982-0565
E-mail: McMullin@santafe.edu

School Of Electronic Engineering
Dublin City University,
Dublin 9,
IRELAND

Voice: +353-1-704-5432
Fax: +353-1-704-5508
E-mail: McMullin@eeng.dcu.ie
Web: http://www.eeng.dcu.ie/~mcmullin

[4] http://cbl.leeds.ac.uk/nikos/tex2html/doc/latex2html/latex2html.html

[5] http://cbl.leeds.ac.uk/nikos/personal.html

[6] http://www.arch.su.edu.au/~peterw/latex/harvard/

[7] http://www.arch.su.edu.au/~peterw/index.html

Retrieval

The resources comprising this report are electronically retrievable, in various formats, via the World Wide Web. The root (directory) URL is:

```
http://www.santafe.edu/sfi/publications/
        Working-Papers/97-01-002/
```

This is also mirrored at:

```
http://www.eeng.dcu.ie/~autonomy/bmcm9702/
```

Either of these URL's will directly access the root node of the HTML format report. Alternatively, an additional filename may be appended, to retrieve one of the other formats, or related resources, as follows:

- The complete set of files making up the HTML format of the article is available in the single compressed archive file `bmcm9702.html.tar.gz` (c. 67 KByte), which can be conveniently downloaded for local and/or offline browsing.
- A printable version of the report is available in postscript format as file `bmcm9702.ps` (c. 206 KByte, 18 pages) or in compressed form as file `bmcm9702.ps.gz` (c. 71 KByte).
- The original LaTeX source of the report is available as file `bmcm9702.tex` (c. 66 KByte) or in compressed form as file `bmcm9702.tex.gz` (c. 22 KByte).
- The BibTeX source for the reference list is available as file `bmcm9702.bib` (c. 4 KByte).
- A complete set of all the files just listed is available as a single compressed archive file `bmcm9702.tar.gz` (c. 162 KByte).
- Finally, the source code distribution of SCL is available in the single compressed archive file `scl-0.04.tar.gz` (c. 32 KByte)