Digital protocells with dynamic size, position, and topology

David H. Ackley
University of New Mexico, Albuquerque, NM 87131
ackley@cs.unm.edu

Abstract

In spatial computational models such as cellular automata (CA), designing mobile objects larger than the CA neighborhood is challenging when the object properties and dynamics are incompletely specified in advance. This paper introduces C211, a two-dimensional digital ‘protocell’ with life-like and potentially useful features, designed for the best-effort asynchronous CA called the Movable Feast Machine (MFM). The protocell consists of an amorphous variable-density ‘cytoplasm’ that uses gossiping to coordinate operations such as cell movements, surrounded by an asymmetric ‘bilayer membrane’ providing some environmental isolation while adapting to cytoplasmic dynamics. C211 was engineered in a new ‘little language’ called SPLAT, which adds discrete 2D spatial pattern transforms to the ulam programming language. SPLAT is expressive enough that minimal code was required, for example, to enable membrane topology changes such as cell splitting and fusion. C211’s cytoplasm maintains internal state but leaves dozens of bits unused per atom, while its membrane is purely stigmergic and stateless—so vast tracts of pristine CA state space remain available for future cellular dynamics, whether engineered, evolved, or both.

Better biology through chemistry

In casual thinking about reality, and in our artificial life designs, we often take ‘agents’ or ‘creatures’ as distinct, given entities, though we know the fine-grained material world makes no ultimately crisp distinctions between the ‘stuff’ of a macroscopic object and that of its environment. Employing such a priori agents can make sense if they are largely stable over durations of interest, and if we are prepared to model all relevant stability violations explicitly. In evolutionary algorithms, and often in ALife models, the ‘agents’ are fixed and static by default, and dynamics like birth, death, and genetic change—plus any environmental interactions—are defined as separate, opaque steps.

This approach is simple and direct, but it risks obscuring, or even obliterating, interesting interactions other than those explicitly modeled. Unexpected interagent or environmental effects may be missed, for example, but also the supposedly opaque steps themselves are inevitably coupled, to some degree, via their joint implementations within some single underlying physics. Reproduction and birth use internal copying but also environmental movements, death takes time, and for a time the dead take space, and so on.

In part to explore such interactions, ALife investigations based on artificial chemistry (Banzhaf and Yamamoto 2015, Dittrich et al. 2001) are good overviews) begin with the underlying physics. Fixed objects and opaque processing steps still exist, but now represent ‘atoms’ or ‘molecules’ and ‘reactions’, out of which the ‘biological’ agents are composed or constructed. In such models (e.g. Sayama 2009, Schmickl et al. 2016, among many), space and time may each be discrete or continuous, open-ended or bounded, uniform or variable, and the chosen space may be 3, 2, or 1D, or even ‘0D’, as in Fontana’s classic stirred reactor.

Mobility in the Movable Feast Machine

For a satisfying artificial chemistry, the artificial physical laws should be motivated by more than just the creatures and interactions to be implemented immediately using the physics. The chemistries reported here are constrained by their implementation on the two-dimensional ‘Movable Feast Machine’ (MFM), an active media computational architecture providing strict indefinite scalability (Ackley and Ackley 2016, Ackley et al. 2013). Given sufficient power, cooling, space, and money to buy hardware, an arbitrarily large MFM computing device could be fabricated without ever encountering any internal engineering limits.

Though employing an active media architecture largely decouples the artificial chemistry from the real-world thermodynamics of power and cooling, it is still significantly constrained by strict indefinite scalability. The ‘simulated virtual space’ within the model must be coextensive—at least above some granularity—with the actual physical space occupied by the hardware. Strict indefinite scalability also implies the computation cannot be globally synchronized, and 100% reliable execution cannot be guaranteed.

Under such circumstances, important computational objects must be prepared to reproduce themselves for robustness, and to move in the virtual and physical spaces of the machine—to escape failing components, or to colonize...
newly-discovered computational real estate, or simply to remain comfortable and effective given incremental movements and changes among one’s neighbors and colleagues.

Though the need for such behaviors can be seen as an engineering consequence of indefinite scalability, the behaviors themselves, of course, are hallmarks of artificial life, prompting [Ackley and Small (2014)] to argue that software-based ALife research has a foundational role to play in the development of indefinitely scalable computing systems. As another basic step in that research and development program, this paper presents the C211 ‘digital protocell’, a collection of data structures and processing mechanisms, running on the MFM, that offer spatial containment, incremental mobility, room for amorphous data storage and processing, and abilities for such objects to split and merge.

Large-object mobility—where ‘large’ means anything notably bigger than the CA neighborhood size—creates inherently conflicting design pressures on whatever plays the role of ‘object’ or ‘agent’ (or ‘component’ in the sense of [Bedau et al. 1998]) in the system. The very idea of an object usually implies some spatial cohesion or compactness among the object’s parts, while mobility implies that one notional object will occupy different spatial locations at different times. But CA sites are fixed in space and discrete in space and time—so in between being compact and cohesive here, and compact and cohesive over there, something has to give.

At least three approaches to large object mobility in CA-like systems are evident in previous ALife and related work:

1. **Global determinism:** The most widely-known CAs are both synchronously updated, so bits can move in and out of a site simultaneously, and globally deterministic, providing complete open-ended predictability.

   The gliders and spaceships of Conway’s ‘Game of Life’ ([Gardner 1970](#) [1971](#) [Adamatzky 2010] and many others) are icons of mobility using this approach; other examples range from the ‘reproduces everything’ machine of [Fredkin 1990](#) to systolic arrays ([Kung 1982](#)).

2. **Object-level synchronisation:** Though synchronisation undeniably eases object cohesion and mobility, global machine synchronisation is neither necessary nor robustly scalable. [Arbib 1966](#) presented a cellular automata with mobility wherein primitive machines assemble into larger mobile objects using “weld points” along their edges.

   Although such welded objects conveniently offer both rigidity and mobility, they depend on scheduling and updating mechanisms that grow increasingly non-local with the sizes of the welded objects.

3. **Asynchronous deformability:** To obtain large mobile objects without global synchronisation (unlike 1), while preserving a constant spacetime bound on local updates (unlike 2), different parts of a large object must move at different moments, so the object’s shape will be deformed, in some fashion, while it moves. Natural biology, of course, is rife with deformable structures; instances of extreme rigidity are rare.

   Previous MFM work, as an ALife example, showed how an object is temporarily distorted while a swapline (Ackley and Ackley 2016) also see Listing 4 below) moves through it—but after the swapline has passed, the object’s shape is restored, except now shifted by one row or column. Swaplines have been used for both large object mobility and reproduction using a ‘quasirigid’ style, as in the ‘2D printer’ example presented in [Ackley 2016](#).

   In a related area, local update rules in the BITPICT system often use temporary deformations during large-object movement (e.g. [Furnas and Qu 2002](#)). Though BITPICT transitions are serial and typically deterministic, many such techniques have analogous forms in the MFM.

   Here again we follow strategy 3, developing a new membrane to surround protocellular ‘cytoplasm’—which is a form of the ‘mob’ elements described in [Ackley and Ackley 2016](#). A mob is naturally diffuse, deformable, and distributed, making it a good candidate for a mobile system. Coordinated decision-making demonstrates how the distributed cytoplasm can compute in a limited way; in this paper we leave open the question of how to perform more complex computations atop mob and membrane.

   To make the discussion concrete, Figure 1 captures three of the C211 protocol’s major behaviors. First, Seed growth into a membrane-enclosed group of 64 Content atoms, using the ‘telomere’ mechanism (Ackley and Ackley 2016) for distributed growth control. Second, the construction and maintenance of a flexible but topologically stable interior/exterior distinction even during content configuration changes, via the InnerMembrane (IM) plus OuterMembrane (OM) mechanism that is a primary technical contribution of this paper. Third, coordinated protocol mobility—with choice of approximate direction and speed—implemented via a customized ‘mob rule’.

   Two more subtle points in Figure 1 will contribute to the discussion later. First, note that every protocol interior site is separated from every orthogonal or diagonal exterior site by IM followed by OM, but the professed ‘bilayer’ membrane is sometimes thicker than two atoms. Second, note how there are empty interior sites even when the protocol is ‘stationary’ (e.g., 150–293 AEPS), but it appears that the protocol’s density decreases when it is moving (1000 AEPS).

**Outline of the paper**

The next two sections introduce the SPLAT programming language with a few motivating examples, then introduce the new membrane itself. Next there is some early data on mobility-induced deformations, plus some anecdotal observations of protocells reacting to environmental stresses, and finally the paper concludes with brief remarks.
Figure 1: Dynamics of protocell growth and mobility, from an initial condition of one Seed atom. By 1 AEPS (Average Events Per Site, pronounced 'eps'), the Seed has ‘sprouted’ into the minimal C211 configuration, consisting of one immature Content atom (Co, in state-dependent shades of green), plus a layer of InnerMembrane (IM, lighter blue) and OuterMembrane (OM, darker blue). The membrane begins to retreat from the high-density content (10 AEPS), creating open sites into which Co atoms divide (25–100 AEPS) until maturing at 64 atoms (150 AEPS). At 293 AEPS the experimenter introduces a Commander (CM, pink) atom which induces a persistent weak bias in the Co motions, and by 1,000 AEPS the protocell is well on its way elsewhere.

**SPLAT the language**

SPLAT, an acronym for ‘Spatial Programming Language, ASCII Text’, was designed with three primary goals:

1. Define plain-text notations for discrete 2D neighborhood transitions, that are constructible and viewable without special software, and are understandable, at least in simple cases, with minimal training; such that

2. The represented neighborhood transitions can be automatically compiled into executable code for an indefinitely scalable computer architecture; and

3. Expressing simple transitions should be simple, and expressing arbitrary transitions should be possible.

To achieve executability, SPLAT source code is processed by a translator—called ‘splattr’—that generates ulam code [Ackley and Ackley, 2016], which in turn is compiled for the indefinitely scalable Movable Feast Machine (MFM) computer architecture [Ackley et al., 2013].

SPLAT defines an execution model and a variety of pattern matching and transformation mechanisms—many of which are touched on below—but it also provides a ‘ripcord’ for arbitrary ulam code injection, allowing the programmer to implement transitions that may prove hard in pure SPLAT. Such code injection is inelegant but powerful, and while we hope future SPLAT development will reduce the ripcord pulling frequency, ulam is good at what it does, and spatial programming is not always the right tool for the job.

To represent spatial patterns in a directly viewable textual form, SPLAT begins by assigning a unique two-dimensional integer coordinate to each byte of a source program text, starting from (0,0) for the first byte of the file with increasing x running to the right and increasing y running down the lines of code. A SPLAT file is interpreted as a quarter plane, with ‘virtual whitespace’ supplied as needed below the last source line and to the right of each.

To ensure that coordinate mapping is unambiguous and visually obvious, the legal character set for SPLAT is a subset of 7-bit ASCII. splatt tr rejects not only modern multi-byte character encodings, but also all ASCII whitespace other than space and newline—including that ancient evil, the TAB byte.

---

1This presentation regretfully presumes the reader has some familiarity with ulam and the MFM.
A SPLAT program modifies that execution model by supplying code sections that provide class metadata, sets of transition rules, and optional ulam data members and methods. A rule set can contain both spatial pattern transition rules, and sentential overrides to modify the default behaviors of the four key methods used in Figure 2:

1. The GIVEN method can force a rule to be abandoned because of inappropriate site content. By default, any site that actually exists in the grid is acceptable.

2. The VOTE method biases a random ‘winning’ site selection for a given key. By default one vote is cast per site.

3. The CHECK method, run after all voting results are available, can also force a rule to be abandoned. By default the rule is abandoned if no votes were cast on the given key.

4. The CHANGE method updates sites based on all available information. By default the key winner is copied.

Some examples
A few small examples can help make all this more intuitive. Consider this complete SPLAT source code for an element called WestGoer, which heads west through empty sites:

```plaintext
function SPLATEVENT(a)  # An event on class a
    for t ∈ a.rulesets do
        for r ∈ t.rules do
            ur ← ∅  # Set of keys in LHS
            for s ∈ r.left do
                Sites s in LHS
            end for
            k ← s.key, ur ← ur ∪ k
            if k.GIVEN(s) or next r  # Failed given
                k.VOTE(s)  # Pick winning site per k
            end if
        end for
        for k ∈ ur do
            k.CHECK(0) or next r  # Failed check
        end for
        for s ∈ r.right do
            k ← s.key
            k.CHANGE(s)
        end for
        return success  # Some rule succeeded
    end for
end function
```

Figure 2: SPLAT event processing pseudocode. See text.

Spatial and sentential forms
At the lexical level, a SPLAT program consists of a sequence of forms, each of which is either ‘spatial’ or ‘sentential’. A spatial form is any consecutive sequence of lines that all begin with a space in column 0. Each spatial form is parsed as a 2D grid containing pattern rules and optional comments.

A sentential form, by contrast, is introduced by a non-space byte in column 0 of a line, and is parsed as a one-dimensional linear sequence of text, similar to many traditional programming languages. A single sentential form may be continued across multiple source lines by beginning each additional line with a ‘.’ byte in column 0. Sentential forms are used for source code organization and declarations, and to inject (more or less raw) ulam code.

The SPLAT execution cycle
SPLAT’s signature feature is its syntax and semantics for spatial pattern transitions. A transition rule is expressed inside a spatial form, and consists of a left-hand side (LHS) and a right-hand side (RHS). Each side is a single connected component based on Moore neighborhoods, delimited in 2D by whitespace or a spatial form edge, and joined into a rule by the ‘->’ operator appearing anywhere between them. The non-blank locations in an LHS or RHS are called sites, while the ASCII character appearing in a site is called a key.

The processing both within and between transition rules is controlled by the SPLAT event execution model (see Figure 2), which is far more structured than the arbitrary object-oriented procedural code used for event processing in ulam.

<table>
<thead>
<tr>
<th>Key</th>
<th>Match on the LHS</th>
<th>Change on the RHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>The unique active atom</td>
<td>Copy active atom to site</td>
</tr>
<tr>
<td>.</td>
<td>Anything</td>
<td>Change nothing</td>
</tr>
<tr>
<td>?</td>
<td>Any non-empty</td>
<td>Change nothing</td>
</tr>
<tr>
<td>_</td>
<td>Only empty</td>
<td>Set site empty</td>
</tr>
</tbody>
</table>

Table 1: SPLAT special keys for spatial rules.

To associate input and output sites, the LHS and RHS of each rule must have exactly the same shape. For example, in line 3 above the LHS and RHS are each a 2x1 rectangle. Also, an LHS cannot sensibly extend more than Manhattan distance four from the @ key, reflecting the size of the underlying MFM event window.
Of course, the horizontal WestGoer patterns can be expressed in single-line linear flows like traditional code, but in NorthGoer a single pattern stretches across lines 3–4:

```
Listing 2: A pattern with height
1 = element NorthGoer
2 \color #ccf
3 == Rules
4 _ @  # If empty above,
5 @ -> _  # swap up
6 @ -> .
7
```

For more customization, the SPLAT programmer can define ‘overrides’ to alter what a key matches and what happens then. Consider the Spread element, for example:

```
Listing 3: A custom GIVEN
1 = element Spread
2 \symmetries all
3 == Rules
4 given s isa Spread
5 _@s -> _@.
6 @ -> .
```

which will move away to an empty site rather than remain next to another Spread. The ‘\symmetries all’ metadata on line 2 causes an orthogonal coordinate transform to be chosen at random before each event, so the line 5 rule can potentially match in any of four directions.

Two last examples show other SPLAT language features:

```
Listing 4: SwapLine
1 = element SwapLine
2 \color #222 \symbol SL
3 == Rules
4 vote s isa SwapLine
5 s@ -> _.  # Thin out
6 @ -> .
7
Listing 5: Zombie
1 = element Zombie
2 \color #666 \symmetries all
3 == Rules
4 given r isa Res
5 given d isa DReg
6 let x = "(r|d|_)
7 x@ -> @@
8 @ -> .
```

The SwapLine rule at lines 8–10 seems to do nothing, but just by succeeding—when either adjacent SL has fallen behind—it blocks the line 12 swap. Note how here VOTE implements an ‘or’ over sites, where GIVEN yields an ‘and’. (See Ackley and Ackley 2016 for more on SwapLine). Finally, the ‘let’ at Zombie line 8 defines key ‘x’ via a boolean expression over other keys: Zombie converts everything it touches—except Res, DReg, and Empty—into more Zombie.

**Membrane design**

Those small examples suggest how SPLAT tries to ‘make the simple simple’. C211—the twelfth protocell version co-developed with SPLAT—is much more complex, but in the end its essence is fairly intuitive.

Over the years this author has explored a variety of ‘membrane-like’ structures—in the MFM and other CA-style architectures—typically coded directly in traditional linear-sentential programming languages. Design and implementation was often difficult and unpleasant, geometric concerns and regularities were often missed or obscured, and even if it worked it often lacked coherence and elegance.

The desire for a better membrane was the proximal impetus for developing SPLAT, and it was designed and evolved in tandem with the C-series membranes. So although it is satisfying that the resulting code is relatively compact and obvious, using only spatial patterns and no internal state, such touches of elegance may or may not occur in other tasks.

The bilayer membrane uses two types, InnerMembrane and OuterMembrane, but their interwined dynamics are lifted into a common base class called QMembrane, leaving only minimal code in their individual definitions:

```
Listing 6: Zombie
1 = element OuterMembrane isa QMembrane
2 \symbol OM
3 \color #45679
4 \symmetries all
```

In C211, QMembrane.splat runs some 154 lines of code, so we omit parts of it here, but see Ackley (2018) for full open-source code, included with the SPLAT translator and its runtime system. QMembrane defines four rule sets—one centered on OuterMembrane and focused on membrane expansion, two centered on InnerMembrane and focused on contraction, and the typical backstop ‘otherwise hold’ rule.

As potential QMembrane designs were explored, symmetries between OM and IM rules emerged, and although the perfect such duality remains elusive, Figure 5 excerpts some of the deeply complementary aspects of the OM and IM code.

A crucial enabler of the membrane design is SPLAT’s voting mechanism, which, among other things, allows estimating local densities by counting sites in complex, rule-specific ways. For example, the ‘run out’ growth rule at lines 33–35 makes configuration changes as in this example:

```
Listing 7: Zombie
1 = element Zombie
2 \color #666 \symmetries all
3 == Rules
4 given r isa Res
5 given d isa DReg
6 let x = "(r|d|_)
7 x@ -> @@
8 @ -> .
```

which moves an existing OM ‘bend’ to increase interior space—but the line 14 check fails unless the external ‘e’ sites are at least 2/3rds empty, while the line 17 check might fail unless sufficient Co are counted in the interior ‘f’ sites. It seems unlikely that these exact rule constraints are absolutely either critical or optimal—but the larger language de-

---

2 Although complex SPLAT rules do become more directly readable as one gets used to a program’s chosen overrides, here we cheat and offer sample transitions that a given rule could produce.
== Rules: OM management (mostly growing) ==
given @ isa OuterMembrane
given o isa OuterMembrane
given q isa QMembrane

... given e : true // Allow dead sites
vote e isa Empty // Count empty sites
cHECK e : $nvotes * 3u <= $nsites * 2u

vote f isa QContent // Count up content
check f : random . oddsOf ($nvotes, 3)

# Growth rules
ee@qiff ..@ ....  # Run out
eooiff ........
eooiff -> ... i ...  # Break out
eo ..
_oif o...
_oif -> @i..  # Punch out
eo ..
ff ..
qiiif ....
o@if -> .i..  # Punch out
tqiiif ....
ff ..

vote r : $curatom is Empty
. || $curatom is OuterMembrane
. || $curatom is InnerMembrane

... given f : true // Allow dead sites
vote f : !($curatom is Empty)
check f : random . oddsOf (3u*$nvotes + 1u, 10u)

# Shrink rules
ee_iqff ..@ ....
eii@off -> ..o.... # Run in
eiiqoff ....

ei ..
_iof i...
_iof -> @o.... # Break in
_iof ..
ei ..

ff ..
qoof ....
i@of -> .o....  # Punch in
qoof ....
ff ..

vote r : $curatom is Empty
. || $curatom is OuterMembrane
. || $curatom is InnerMembrane

... A non-membrane site is defined to be an interior site if it is adjacent to an IM or another interior site, and likewise for exterior sites and OM. A collection of sites is called membrane consistent if every non-membrane site is uniquely labeled as interior or exterior. All the rules in Figure 3 (including 'cave out', not exemplified here) are membrane invariant, giving membrane-consistent outputs from membrane-consistent inputs. Disjoint sets of interior sites define distinct protocells, but with the membrane invariant, global protocell identities can be changed locally. One IM-centered rule does membrane-invariant fission, making two from one:

A non-membrane site is defined to be an interior site if it is adjacent to an IM or another interior site, and likewise for exterior sites and OM. A collection of sites is called membrane consistent if every non-membrane site is uniquely labeled as interior or exterior. All the rules in Figure 3 (including 'cave out', not exemplified here) are membrane invariant, giving membrane-consistent outputs from membrane-consistent inputs. Disjoint sets of interior sites define distinct protocells, but with the membrane invariant, global protocell identities can be changed locally. One IM-centered rule does membrane-invariant fission, making two from one:

while an analogous OM-centered rule implements fusion:

(See Figure 3 for examples of these rules in action.)
If global deterministic execution was absolutely guaranteed, we could simply require a membrane-consistent initial condition and declare job well done, but in an indefinitely-scalable computational architecture, it is necessary to ask the hard question: What should we do if, despite everything, the membrane is inconsistent? Most of the QMembrane code omitted above is concerned with self stabilizing the membrane—attempting to repair it safely if plausible, or to destroy it cleanly if not, when facing a membrane-inconsistent state. More work is needed, and many interesting challenges arise, but they run too far afield for us here.

Cytoplasm design

As noted earlier, the C211 cytoplasm is similar to the ulam Mob element presented in [Ackley and Ackley 2016], except it’s a reimplementation in SPLAT—and less than a lovely one at that—so we offer only brief comments here.

Putting a membrane around mob Content had at least two design consequences. First, although the membrane retreats from high-density Content, its presence does help ‘corral’ the mob, so the ‘statistical gravity’ used in Ackley and Ackley (2016) is less needed. Second, the original mob cared only about its heading direction, empty sites, and mob sites; anything else encountered was, in effect, an obstacle. Within a membrane, though, that caused a race between the desired effect: the membrane leading edge advancing away from oncoming Content, and an undesired effect: the mob rule increasingly swapping with empty sites laterally or behind as the density ahead increased. Fitful attempts to adjust rate constants proved futile, but a new Content rule encouraging adjacency to IM in the heading direction was effective.

Asynchronous deformability quantified

Though these protocells consist primarily of OM, IM, and Co atoms (plus empty sites), additional elements exist as well. For example, if a Co encounters a “Commander” atom, as after 293 AEPS in Figure 1 it consumes it, then adopts a random heading direction and speed, which it gossip to neighboring Co. A StopCommander atom ends movement similarly. Content atoms use stochastic timers to emit such signaling atoms spontaneously, generating random protocell movements and pauses somewhat akin to the ‘run and tumble’ dynamics of natural bacteria such as E. coli [Airola et al. 2013] e.g., details its signaling and switching mechanisms).

As discussed in the introduction, in this architecture, object motion requires object deformation, so one might hypothesize that faster motion might imply more deformation. We conducted a small experiment to evaluate that, running the ‘Run and Tumble’ dynamics and computing protocell volume and distance traveled every 500 AEPS. As seen in Figure 4 the results are noisy, but a moderately strong positive correlation between volume and speed is evident.
Figure 5: Context-induced protocell topology changes, extracted from three runs and displayed using relative AEPS. (Top row) A protocell rushing west hits an obstacle. At +2,505 the stretched lower membrane has fissioned and the parts are shrinking back, but too late: A second fission at +2,840 leaves this run with three stories going forward. (Middle row) By contrast, this far slower westbound protocell, though also bent by the wall, eventually takes a northerly route and remains intact. (Bottom row) The green-Content protocell, motionless on an edge, is squeezed by the protocell from the east, and by +2,000 they are sharing a vertical segment of 0M. At +2,130 the two membranes fuse, and a double-sized yellow-green protocell rapidly consolidates.

Acknowledgments

Elena S. Ackley implemented the ulam compiler underlying SPLAT. Some early SPLAT and membrane development took place while the author was visiting the Computer Science Department at Brown University. This work was supported in part by grant VSUNM201401 from VanDyke Software.

References


