Building a survivable protocell for a corrosive digital environment

David H. Ackley

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

Abstract

A computer produces outputs from inputs, and to do so reliably, its internal noise and variability must be managed effectively. Traditional computer architecture requires *hardware determinism*, but such perfect repeatability is increasingly incompatible with large-scale and real-world systems. Natural living systems, without the luxury of deterministic hardware, manage variability across the computational stack—and using such principles, *soft artificial life* offers a route to much larger and safer manufactured computers. This paper describes the engineering development of *C214*, a next-generation self-constructing digital protocell. *C214* struggles to survive in a challenging environment that, while not literally malicious, goes well beyond merely non-deterministic to deliberately destructive. Improved self-repair mechanisms, as well as active defenses in depth, give the new cell's membrane a median survival time more than ten times greater than that of the earlier *C211*. A new grid-based cytoplasm is also presented, standing to offer a more stable environment for future layers of the 'living computation stack', and some basic cellular software engineering techniques are highlighted.

Soft artificial life and society

The ALIFE 2019 conference theme asks us: "How can Artificial Life help solve societal challenges?" Indeed, while making even software-based alife is a grand challenge on its own—and the results can be fascinating purely as objects of study—this author agrees that alife is also called to help address urgent problems in human technological society.

As an example, the author's research and development program seeks to show that 'soft' alife, deployed on a new computer architecture, offers a radical but coherent solution to one of technological society's 'wicked problems': The inability to make computer systems securable and robust.

This paper is a progress report for ongoing work with "digital protocells"—spatially-extended software constructs designed for the indefinitely scalable "Movable Feast Machine" (MFM) computer architecture (see, e.g., Ackley, 2013). These protocell designs, as well as performing life-like tasks such as growth and repair, are intended to serve as foundational components for future "multicellular" computations that are *inherently* robust—able to survive, even flourish, without global hardware determinism.

While the original protocells presented in Ackley (2018) operated in an otherwise empty universe, these new *C214* cells struggle under the "`DReg` and `Res`" (`DR&R`) computational regime—in which any atom adjacent to a `DReg` atom may be erased at random without notice, no matter the victim's possible importance in any ongoing computation.

In the rest of this introduction, we provide motivation and context for the MFM architecture and the soft alife approach to robust-first computing, then outline the rest of the paper.

Computer scalability and securability

Since the 1940s, manufactured digital computers have revolutionized many areas of human society. Untold billions of units are now deployed in myriad uses and physical forms—but virtually all of them presume *global hardware determinism*: The hardware is to produce perfectably repeatable software execution, over the entire computer, however much time or available memory the software consumes. Any potential failures are to be eliminated *during system design*, for example by error-correcting codes or other fault tolerance.

The resulting determinism guarantee makes computations easier to implement, because application software need not worry about processor errors, or check its work, or confirm that values in memory are still good. Deterministic execution is a powerful and clean separation of concerns between hardware and software—but *guaranteeing* it becomes just unaffordable as computational systems grow. As the supercomputer community already knows (Cappello et al., 2009), for example, if you run enough hardware long enough, it *will* deliver undetected errors to software—and. . . Then What?

Because of software's relentless focus on correctness and efficiency only, computing today has virtually no answer to that question. When, for whatever reason, something *does* go wrong *in a delivered system*, we simply have no idea how badly the ongoing computation will be damaged; we have no general way to estimate *how wrong* its output is likely to be.

Another—at present much worse—problem is that even when some task does fit comfortably within global hardware determinism's limited 'globe', software's systemic lack of redundancy also means it has *no structural defense* against

unexpected changes caused not by hardware failures but by malicious attacks. While redundancy, recomputation, and cross-checking are no *guarantee* of 'cybersecurity', if applied across the computational stack they can make crafting attacks much more expensive and their payoffs much more modest. By deploying redundancy effectively, we can also use *statistical* reasoning to estimate damages, long after our *logical* reasoning has collapsed due to perfection lost.

If deterministic execution gives way to indefinitely scalable *best-effort computing*, the 'exotic' techniques of artificial chemistry and soft alife—spatial processing and local memory, self-organization and repair, mobile and reproducing structures, etc.—will eventually become computation as usual. Getting there will take much work, but facing the absymal state of computer security given traditional architecture, this author contends that society sorely needs artificial life researchers to help accelerate that transition.

Outline of the paper

The next section briefly places the present work into the context of some existing work, in computation broadly, and artificial life more specifically. The following sections summarize the goals and mechanics of the DR&R physics, then present the new *C214* protocell, focusing on its design improvements over the *C211* cell previously reported Ackley (2018), and then offer data on the relative impacts of the improvements. A short discussion and conclusions follow.

Related Work

Though this project uses a novel architecture, discussed further below, there is of course much work—both in and out of alife—employing similar goals, concepts, or mechanisms.

For example, work in distributed systems routinely handles non-deterministic execution and many kinds of errors, with decades of empirical and theoretical results (e.g., Saltzer et al., 1984; Clement et al., 2009). Self-stabilizing systems (e.g. Dijkstra, 1974; Dolev, 2000) are also related, as is work in spatial and amorphous computing (e.g., Abelson et al., 2000; Orhai and Black, 2012, is very relevant). One view is the present work applies such distributed systems principles to traditionally 'single host' computations.

Also, although the spatialization and other details differ, the goal of using programmable soft alife to engineer useful computations is shared between the present effort and alife work such as 'chemical networking protocols' built with lovely *Fraglets* language (e.g., Meyer and Tschudin, 2009).

Indefinitely scalable artificial chemistry

Traditional computer architecture specifies dedicated and highly-differentiated spatial structures for CPU, RAM, their interconnections and input and output channels, and that's it. Though there are bigger and smaller CPUs and other components, they are all necessarily limited in size—RAM, for example, cannot extend too far from CPU, or access time suffers. About systems much bigger than those limits, traditional computer architecture has little to say.

The trap of deterministic cellular automata That fundamental boundedness stands in stark contrast to *cellular automata* (CA) computational models. Their dimensionalities and details vary, but CAs virtually always specify *spatially isotropic processing*: At some level of description, the *same* function is performed at *all* locations in computational space. Though any CA instance must be finite, and spatial isotropy may be violated at its edges, the powerful implication is that *any finite size* could be chosen.

But CA models that require deterministic execution lack indefinite scalability and violate that assumption. As architectures, deterministic CAs are doomed to lose exactly that spatial unboundedness that made them so appealing at first. Much has been learned from them, but for truly large-scale computations, deterministic cellular automata are a trap. They are climbing a tree to get to the stars.

Artificial chemistries and the MFM Alife work based on *artificial chemistry* (Banzhaf and Yamamoto, 2015, is a good overview) begin with a physics model, where the primitive objects and processes represent 'atoms' or 'molecules' and 'reactions', from which 'biological' agents are constructed.

Unlike CAs more generally, artificial chemistries typically avoid the trap of determinism—for example, the exact order of reactions is usually random—but other assumptions can limit the scalability as well. Models that critically depend on floating-point accuracy are problematic, for example. Similarly, some dimensional assumptions—such as Fontana (1992)'s classic '0D' well-stirred reactor—must be reexamined as well: Bigger beakers take longer to stir well.

The present research program focuses on chemistries that are implementable on the 2D 'Movable Feast Machine' (MFM) (e.g. Ackley and Ackley, 2016; Ackley et al., 2013). The MFM is designed first and foremost to be an *indefinitely scalable computer architecture*, and it jettisons everything that might prevent such open-ended scalability. Though it uses discrete sites and local transition rules like typical CAs, the MFM offers only best-effort reliability and guarantees essentially nothing about the cellular automata site update orders, or even the rates of receiving update events over time.

Those assumptions and others make MFM programming much more complicated than traditional software—but that added effort up front stands to pay off *indefinitely*, as the underlying hardware expands and improves.

An energetic, corrosive environment

The basic 'laws of physics' examined in this paper begin with DReg and Res, two of the oldest MFM elements, first reported in Ackley and Cannon (2011), and considered at length in Ackley (2013). DReg and Res form a homeostatic mechanism designed to help manage free space and con-

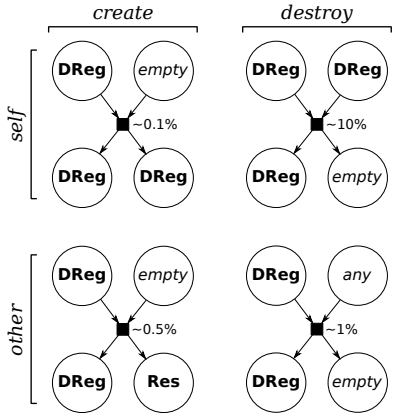


Figure 1: DReg reactions overview, with inputs *(circles with outbound arrows)*, outputs *(circles with inbound arrows)*, and approximate reaction probabilities *(small black boxes)*. DReg may create Res or (more) DReg, or destroy DReg or *any other type of atom*. Empty sites are explicitly modeled in these pseudo-chemical reactions because the Movable Feast Machine is based on *conservation of space* rather than of *matter*. See text. (From Ackley, 2013).

trol crowding. Res (R) represents a 'generalized **R**esource' element that does nothing, on its own, except diffuse randomly into empty sites—but it can be employed or consumed however a computation wishes. By contrast, DReg (DR)—a '**D**ynamic **Reg**ulator' element—randomly creates Res (and more DReg), while also randomly consuming *everything*—including, in particular, pieces of whatever other computation is running. Figure 1 depicts the DReg reactions.

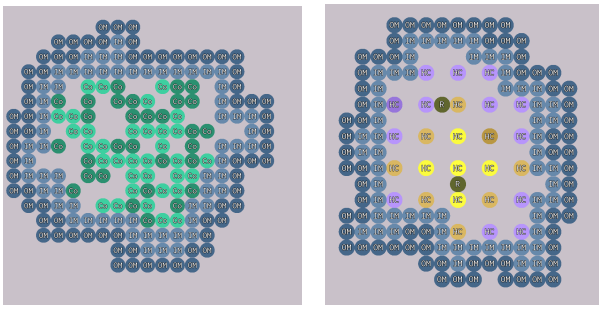
It's important to note that this DR&R mechanism is not inherent in or enforced by the underlying MFM architecture—it's purely 'opt-in' from the point of view of the element programmer. We don't have to include DR&R in the 'table of elements' of our program, and if we do, we can easily write state transition code that simply erases every DReg atom that our element encounters. Similarly, we don't have to wait for a Res to create new atoms if we don't want to.

But if we *choose* to opt-in to the DR&R execution environment, we play along with its ground rules:

- Don't create or destroy DR; only DR itself does that.
- Don't grow unless we can consume R to do so. And,
- DR creates R 'de novo', but we can create R only if we destroy something previously made from R.

The current protocell violates DR&R rules in one major way: The membrane grows and shrinks as it pleases, without consuming or shedding R. But the risk of runaway growth is minimized by the membrane clinging to the cytoplasm—which mostly *does* obey DR&R rules—and by the previously-developed membrane internal consistency checks.

If our computation can persist and make progress while co-existing with DReg, we will have a limited but legitimate basis for calling it 'robust'. The goal of the present work is a simple but robust protocell worthy of the name.



(a) *C211* (b) *C214*

Figure 2: Sample *C211* and *C214* protocells. See text.

A tough protocell for a harsh environment

The original *C211* protocell, and the 'SPLAT' spatial programming language used to implement it, both debuted at last year's Artificial Life conference (Ackley, 2018). *C211* provided a range of useful 'cell-like' properties such as spatial isolation, mobility, and fission and fusion. It has a two-layer 'membrane' made of InnerMembrane ('atomic symbol' IM) and OuterMembrane (OM), plus an amorphous 'cytoplasm' element called Content (Co). It had some abilities to repair damaged membranes, but it was not explicitly designed for nor extensively tested in the DR&R regime.

The new *C214* protocell addresses that limitation. *C214* uses updated versions of IM and OM, plus two new cytoplasm elements called 'HardCyt' (HC) and 'SoftCyt' (SC). Figure 2 provides samples of the old and new protocells, and the rest of this section discusses the changes between them.

A more robust membrane for controlled isolation

***C212* membrane development**

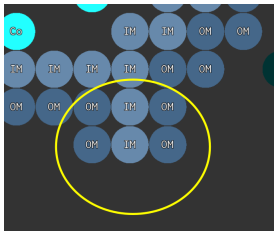


Figure 3: An unmatched *C211* state.

Initial tests of *C211* cultured in DReg/Res revealed, among other things, some uncovered edge cases in its spatial transition rules. For example, a configuration such as in Figure 3 proved to be a 'pinning state' because, as it happened, no rules matched that shape. That didn't matter originally, because such an 'exposed IM' state cannot arise during error-free growth of a *C211* in an empty universe—but given DReg's random depredations, such frozen shapes do arise. The rest of the *C211* membrane continues to move around it, but eventually further DReg damage breaches the membrane entirely.

This lacuna was handled via a new SPLAT rule for IM:

```
160 | == Rules: IM management (shrink and die)
161 | given @ isa InnerMembrane
162 | given i isa InnerMembrane

165 | given c isa OuterMembrane

168 | given o isa OuterMembrane
169 | let x = i|o
```

```
182 |   ___       ...
183 |   c@c  ->   .c.      # Cap off
184 |   oxo       ...
```

(where the specific line numbers refer to the *C214* version of `QMembrane.splat`, which incorporates these changes). The excerpts in lines 161–169 declare a variety of 'keycodes' for matching purposes; more than one keycode is sometimes declared for the same type (e.g., lines 165 and 168) to make the state transition rule more specific (discussed below.)

The left-hand side (LHS) of the spatial pattern rule at lines 182–184 matches states including the yellow-circled one in Figure 3: Three empty sites (line 182), then an `OM`, `IM`, `OM` sequence (line 183, with the **`@`** keycode defining the center of the event), then finally two `OM`s with either an `IM` or and `OM` between them (line 184).

When that pattern matches (as it could, 180° rotated, in Figure 3), the right-hand side (RHS) of the spatial pattern rule is performed. On the RHS the keycode **`.`** means do nothing, while the keycode **`c`** means "write a copy of the 'winning' **`c`** chosen from the LHS." Both of the **`c`**s on the LHS must have matched to reach this RHS, so one of them is chosen at random and copied to the event center, with the net result is that the center `IM` becomes a copy of an adjacent `OM`, and the *membrane invariant* (see Ackley, 2018) is restored.

One might wonder why that rule was chosen, rather than a simpler one such as:

```
    ___       ...
    o@o  ->   .o.      # Cap off
    oxo       ...
```

which would also restore the membrane invariant in cases such as Figure 3. The more complex rule ensures that the copied `OM` is from an immediately adjacent site, while with the latter rule any of the four LHS **`o`**s might have been chosen. Although the result is the same if all `OM` instances are identical, in a complex physics the atoms involved may actually be subclasses of `OuterMembrane` possessing additional state that differentiates them.

Minimizing spatial damage Using the rule as shown ensures that—should an 'emergency repair' to the membrane invariant be necessary—it will be done in a *least spatially damaging* way, by copying from a nearest neighbor of the appropriate type, chosen at random to preserve spatial isotropy. Such nearest-neighbor copying may or may not be what some higher-level computation might want—but it is defensible as the best choice to make absent any other information, purely on the basis of geometry—and we suggest that this is an example of what *best-effort spatial software engineering* looks like (but see Section Selective permeability via virtual stigmergy below for additional discussion).

A few other discovered corner cases were handled similarly. For example, although *C211* had a simple rule to correct a missing `OM`:

```
  8 |given @ isa OuterMembrane
  9 |given i isa InnerMembrane
```

```
81 |   i_        .@
82 |   i@   ->   ..     # Square off (outer)
```

which copies the **`@`** `OM` into an inappropriately empty adjacent site, a somewhat similar rule for `IM` needed to be added:

```
84 |   _i        i.
85 |   i@   ->   ..     # Square corner (inner)
```

The result of these membrane self-repair modifications was dubbed *C212*. Its `DR&R` median survival time (tested below) was a clear improvement over *C211*, but one could certainly hope for better still, considering that each of these protocells represents a significant investment in construction time and space, possibly embodying considerable state information of use to some higher-level computation.

***C213* membrane development** How can membrane survivability be improved further, after the locally-detectable membrane self-repairs are done? One might add a third '`OuterOuterMembrane`' layer, for example. That would allow more time to repair `DReg`-induced damage before the membrane was completely breached, but it would also incur significant area costs as well as a major codebase redesign.

A more feasible alternative holds that the 'real problem' is the `DReg` itself. The `DR&R` regime makes it a sin to kill a `DReg`, but says nothing about trying to *keep them away* from the membrane. To that end, the rule shown in Figure 4 was developed. The spatial rule at lines 39–40 divides the world into three categories: **`D`**s, **`X`**s, and the `OM` itself (**`@`**). **`D`**, defined purely in SPLAT, is a vote for a `DReg`. While SPLAT **`given`** declarations introduce constraints that *all* must be true for a rule LHS to match, **`vote`** declarations, by default, will succeed if *any* LHS appearance of the given keycode receives more than zero votes. The rule in Figure 4 will match if one or more of the three **`D`**s is a `DR`, at which point the expression **`$D.$winsn`** will represent a *site number* containing a `DR`.

The **`vote X`** declaration (at lines 30–35) uses a block statement, in {}'s, using *ulam* code (Ackley and Ackley, 2016) to express a relatively complex weighted voting criterion: Any **`X`** site containing a `DR` or any subclass of `QMembrane` (which includes both `IM` and `OM`), receives zero votes and so cannot be selected; otherwise votes are assigned based on the *squared distance* from the site being voted upon (**`$cursn`**) to the event center (**`@`**). If the **`X`** in line 39 matched, for example, that site, at (-2,-1), would receive five votes ($(-2)^2+(-1)^2$), while the leftmost **`X`** in line 40, at (-3,0), would score nine.

If the rule overall matches—meaning at least one **`X`** received votes and at least one **`D`** matched a `DR`—the RHS changes are performed. A limitation of the SPLAT spatial rule syntax is that it has no direct way to express a transformation like "Swap *whichever* sites won the **`D`** and **`X`** votes", so that desire is expressed sententially instead, via

```
 7 | == Rules: OM management Part 1 (miscellaneous business)
 8 | given @ isa OuterMembrane

28 | # Fight DReg
29 | vote D isa DReg
30 | vote X {
31 | .  return ($curatom is DReg             // Swapping DReg inward doesn't help
32 | .          || $curatom is QMembrane)    // and moving membrane could be disruptive
33 | .       ? 0u                            // So no votes for them
34 | .       : (Votes) ew.getCoord($cursn).euclideanSquaredLength(); // Else farther is better
35 | .}
36 |
37 | change D { ew.swap($X.$winsn,$D.$winsn); }
38 |
39 |     XD       ..
40 |    XXD@ -> D...
41 |     XD       ..
```

Figure 4: A rule to push DReg away from the membrane, excerpted from the *C214* QMembrane.splat. See text.

the **change D** declaration at line 37.[1]

Note this 'DReg-fighting' rule is quite aggressive. It's willing to swap a DR with *anything* suitably farther away that isn't a cell membrane or already a DR—and that absolutely could disrupt some significant structure that happened to be near a protocell's outer membrane. Wearing a traditional deterministic 'top-down programming' hat, one would want to ask if that could actually happen, and handle it ahead of time somehow—but again, from a best-effort, bottom-up, spatialized point of view, that isn't strictly necessary. If the need is great enough, from the point of view of whoever is having the event (here, OM, with the potential destruction of its protocell at stake), that is sufficient grounds for action, full stop. If some other structure thereby gets disrupted, well then it's that structure's fault for being too close to OM and DR.

Overall, the *C213* membrane is identical to *C212*'s but for that single added rule, and it makes a dramatic improvement in membrane lifetime, as discussed below.

***C214* membrane development** Pushing DReg is a great help, but it is not a panacea for at least two reasons:

1. Each event happens with a randomly-chosen symmetry. A DReg-pushing opportunity may be missed because the OM was looking in the wrong direction.
2. The MFM makes no guarantees about event delivery order. It's unlikely but possible that a DReg could approach and burn all the way through a membrane, before the nearby membrane atoms got even one chance to react.

Bad luck can *never* be completely eliminated—and that is almost a mantra for best-effort computing—but it can be minimized statistically. And perhaps the most fundamental way to swing the statistics your way is to *control more space*—which is what the *C213* DReg-pushing rule tries to do. The *C214* membrane, in turn, moves the battlefield even farther from the protocell, by deploying 'cilia' (Ci), which in this case are free-floating individual atoms, rather than literally hair-like structures. Unlike all other protocell components, the cilia live outside the outer membrane, but they remain near and in service to OM. Ci are created by transmuting available R, using an OM rule like:

```
# Deploy cilia
given Z : !($curatom is Cilium ||
.           $curatom is QMembrane)
vote Z isa Res
change Z {ew[$Z.$winsn]=Cilium.instanceof;}

   ZZZ      ...
   ZZZ@ -> Z...
   ZZZ      ...
```

which combines **given**, **vote**, and **change** declarations with a spatial pattern to express essentially this transition: "If there are no Ci, IM, or OM nearby, but there is at least one R, change such an R into a default instance of Ci."

A Cilium atom, in turn, has a variety of rules to push away DR, referencing a nearby OM to decide which direction is "away", as in these excerpts from Cilium.splat:

```
10 | given @ isa Cilium

22 | # Push DReg
23 |
24 | vote o isa OuterMembrane
25 | vote d isa DReg
26 |
27 | change d { ew.swap($_.$winsn,$d.$winsn); }
28 |
29 |  _ddd@oooo  -> .d.......

37 |     @oo  ->   ...
38 |    _doo      d...
```

where keycodes **o** and **d** use multiple sites and voting to help the rule match more often. Ci also have rules to diffuse near

[1] It is (perhaps unfortunately) necessary that **D** appear *somewhere* in the RHS to have any effect, but its specific location is ignored when an explicit **change D** declaration is provided.

the membrane, and to set their own site to empty (E) if no OM can be located—and such DR → R → Ci → E reaction chains are one way that R are 'metabolically consumed' by protocell operations. Overall, Ci is the signature addition differentiating *C213*'s membrane from *C214*'s, and experimental data shows that deploying Ci produced a qualitative jump in the membrane survival time, as discussed below. Figure 5 depicts a small *C214* protocell, with a developing cloud of Ci around it.

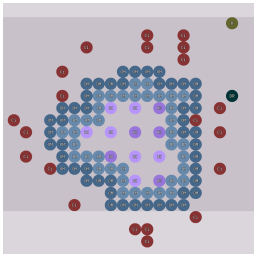


Figure 5: A young *C214* protocell developing cilia.

Although the Ci were developed, initially, solely to provide defense-in-depth against DR, they have since proved useful for other purposes. For example, some protocell operations consume R to obey the DR&R regime—and just as Ci can push DR away, these excerpts from Cilium.splat show how they can also, if deemed appropriate, pull R in:

```
6  | == Data members
7  | u Bool mPullRes = true;

62 | == Rules (Pull res)
63 | given @ isa Cilium : $self.mPullRes
64 |
65 | vote o isa OuterMembrane
66 |
67 | vote r isa Res
68 | change r { ew.swap($_.$winsn,$r.$winsn); }
69 |
70 |  rrr@_oo     -> ....r..
71 |
72 |  rrr_@oooo   -> .r.......
```

where the ': $self.mPullRes' **given** condition on line 63 depends on the value of the data member declared (in *ulam* code) at line 7. Ci's mPullRes defaults to true, but after that it is turned off and on by an additional *C214* OM rule—and how that rule knows whether another nearby R is likely to be a help or a hindrance, at the moment, is discussed in Section Selective permeability via virtual stigmergy, below.

In the near future we expect Ci will gain yet more useful abilities, such as assisting with protocell mobility and perhaps various types of environmental sensing. Its notable immediate utility during *C214* development was a reminder of yet another bottom-up mantra: Space is the place.

A quieter cytoplasm for bounded centralization

The protocell membrane is a crucial component, but of course a container with nothing to contain is only half a story. A version of the Mob element (originally discussed in Ackley and Ackley, 2015), formed the protocell interior 'cytoplasm' for *C211*. That Mob-like element—called Content (Co)—performed controlled growth from a seed, was collectively mobile, and used gossiping to disseminate movement commands. Those movement commands could originate from anywhere inside the cell, because Co was completely also decentralized—which was satisfying from a first-principles robustness and 'bottom-up purist' point of view, but it also proved a challenge when it came to adding further functionality and programmability to the cell.

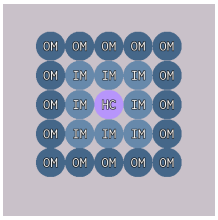


Figure 6: The *de novo C214* configuration, created by a Seed atom.

With that experience in mind, the new cytoplasm design goes a different way. Rather than an amorphous and centerless cloud akin to Mob, the *C214* cytoplasm is basically a semi-rigid grid spaced on two site centers (see Figures 2b and 5), more like the Router elements discussed in Ackley (2016). The grid is constructed from atoms of HardCyt (HC), which grow opportunistically by R transmutation starting from a single HC created (in the *de novo* case) by a Seed atom, which sprouts, as its only transition, into the configuration shown in Figure 6.

Once created, HC atoms do not move—although they can 'melt' into movable SoftCyt (SC) atoms—and HC's relative rigidity underpins a cell-spanning local coordinate system. Each HC works to localize itself within the bounding box of all HC inside the membrane, by maintaining a four number array, as seen in these excerpts from HardCyt.splat:

```
17 | == Data members

19 | u typedef Distance DistanceArray[4];
20 | u DistanceArray mDistances; // WNSE
```

where Distance is defined as a four-bit unsigned integer with a maximum value of 15. These distances are updated, during most HC events, by observing either another HC or a membrane atom in a randomly-chosen direction. When we (as the HC having an event) see a membrane atom, that is (the current) ground truth, and we set our distance in that direction to zero. When we see another HC, we set our distance in that direction to one more than theirs (though saturating at 15), and we set our distance in one of the two orthogonal directions to the max of their distance and ours.

If the HC happen to fill an axis-aligned rectangular shape, this process quickly finds consistent positions for all. More commonly, HC estimates partially settle and then jitter, depending on whether an HC listens to its neighbors or trusts its own eyes. But, in another signature of best-effort computing, although position estimates may be unstable or wrong, other *C214* processes use them *as if* they are correct.

In particular, once the estimated protocell volume has reached a minimum level, HC that find themselves near the center begin competing to be the 'leader' of the cell. Once some such HC is strong enough, in one fell event it declares itself the leader, and begins suppressing the neighboring HC. After a leader has arisen, an estimated distance-from-leader gradient forms—in addition to the ongoing position estimates—across all the HC. With leader competitions spa-

tially limited in this way, we virtually never see multiple separate leaders rising within a connected HC pool.

Having a single leader—like having a single queen bee—is a great help for coordinating protocell actions, and we look forward to fleshing out such uses soon. Of course, a single leader is also a single point of failure, but since the hierarchy's stability depends on the leader (like the queen) actively suppressing competitors, if the leader *is* somehow lost, central competition soon resumes and another rises.

Evaluation of membrane survival times

Here we present the results of a small experiment to assess the effects of the membrane changes discussed above.

An experimental fixture To evaluate survival time changes due to the membrane modifications from *C211* to *C214*, a new `Alien Explosion` (`AX`) element was created. `AX` inherits from `QContent`, so it is recognized as 'self' by `IM`, but its behavior is simple and drastic: If an `AX` ever finds itself next to anything *other* than a `QContent` or an `Empty`, it erases itself after 'triggering' all nearby `AX` to do the same.

In effect, any contiguous cluster of `AX` vanishes almost immediately once any member of the cluster encounters an 'alien' atom. Using the '`mfms`' simulator's command line arguments '`--halt-if-extinct AX`' and '`--haltafteraeps 500000`', the time that a membrane was breached, up to a maximum of 500KAEPS[2], can be assessed with good accuracy.

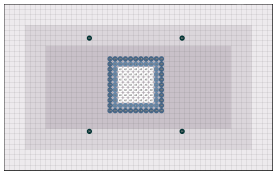


Figure 7: The membrane survivability experimental fixture is a single size 'H' tile (52×32 active sites), with 49 `AX` (*white*), 4 `DR` (*black*), 32 `IM` (*light blue-green*) and 40 `OM` (*dark blue-green*) of the particular membrane being tested. See text.

Figure 7 shows the test fixture used in the experiment. A short distance from a membrane-enclosed block of `AX`, four `DR` are positioned, which gradually populate the extracellular environment with a mix of `DR` and `R`, as discussed above.

Results Membranes of the *C211–C214* protocells were tested 25 times each, and the membrane failure times recorded. Figure 8 presents all the data, with median survival times for each membrane called out.

Over the trials the original *C211* membrane rarely survived for 100KAEPS and never reached 200KAEPS; *C212* did notably better in most cases but never reached 200KAEPS either. *C213*, on the other hand, exceeded 200KAEPS over 30% of the time and once even reached the 500KAEPS test limit. Finally, *C214*'s cilia-waving monster membrane exceeded 500KAEPS on more than 50% of runs.

[2] 1 KAEPS is an average of 1,000 events per site.

At present, longer runs remain to be performed, so no credible estimate of *C214*'s unbounded median membrane survival time is yet known. Of course death eventually comes to us all, and the details remain to be developed, but intuitively at least, the *C214* membrane *feels* survivable enough to support protocell growth and maturation and a productive computational lifetime, before that happens.

Cellular software engineering

It is unusual to see much real code in an alife paper, but programming for an indefinitely scalable machine is different and more challenging than for traditional architecture. Code matters. Programmers think in code, and the metaphoric and literal *shape* of code subtly but strongly influences what future developmental directions will seem feasible. Here we focus briefly on indefinitely scalable software engineering.

'Metabolic' regulation *C211* used a 'telomere' counting mechanism to decide when its `Co` cytoplasm should stop growing, where *C214* now uses a combination of '`R` metabolism'—as dictated by the `DR&R` regime—plus cytoplasmic volume estimation via the bounding box described above. If the cell seems big enough, `HC` production shuts down and `R` import (discussed below) slows. Though this mechanism does not strictly conserve energy or matter or anything, it is quite effective, and—unlike the internally-facing telomere mechanism—it automatically adjusts, at least crudely, to a changing environment.

Selective permeability via virtual stigmergy *C214*, under the `DR&R` rules, grows by importing `R` into the cell. At first we used an `IM` data member to specify an importable element type—but that was limited to one type, and it was unclear how or when `IM` should reset that type, and it consumed 16 precious state bits. A much better solution was to have `QContent` specify a function '`virtual Bool shouldImport(Atom)`' and have the `IM` call it *during its event* to determine if a given atom should be imported.

This way, subclasses can easily customize `shouldImport` to their needs, and it costs *zero* state bits in `IM`, and it's never obsolete. It's so nice that a `shouldExport` method was also defined, and *C214*'s `IM` can actually import and export simultaneously in a single event, using only one SPLAT rule.

This shows a third way to handle context, between defining explicit state to internalize it, and simply ignoring it like the repair rules in Section Minimizing spatial damage. The key to it is *spatiotemporal contiguity*: Having `QContent` reliably close enough that `IM` can run code on it during an event. `Cilium`, by contrast, uses a stateful `mPullRes` data member, despite its problems, to move freely as a forward fighter.

Towards living computation for society

Obviously, this is all just a beginning. There is a tremendous amount of work to do before we can demonstrate robust con-

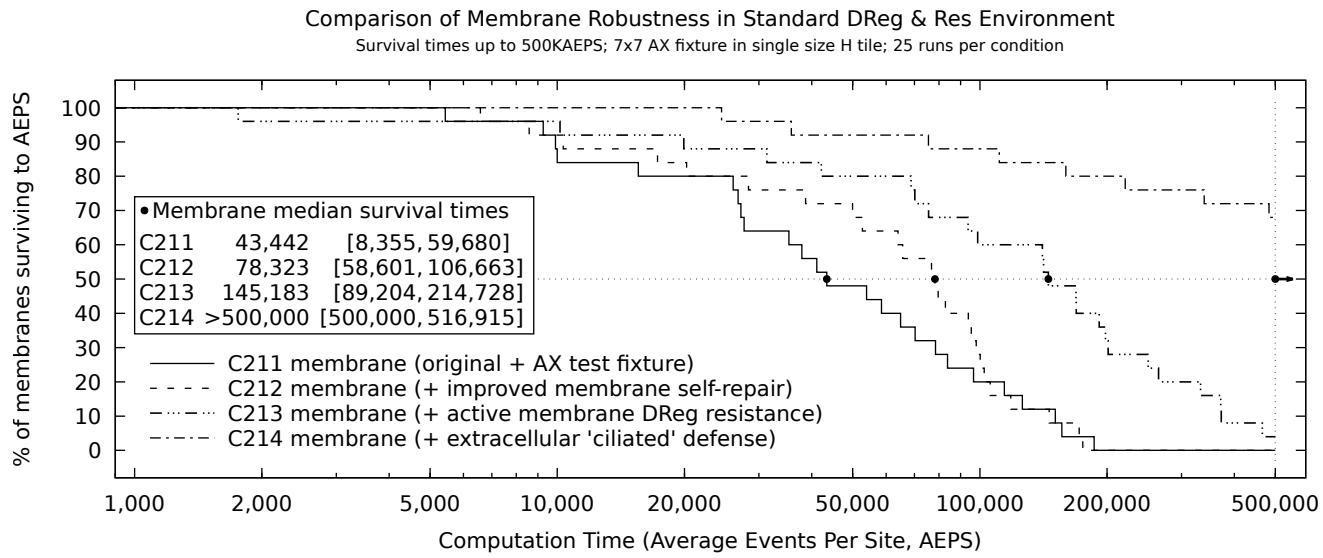


Figure 8: Survival time distributions of the four tested membranes, with medians and their 95% confidence intervals. See text.

crete system control using these indefinitely scalable, living computation methods. In at least one way, in fact, the *C214* protocell takes a step backwards: Its mobility is quite limited compared to *C211*. Some blame goes to the rigidity of HC compared to Co, but, in retrospect obviously, it is also due simply to the existence of DReg and Res, compared to the utterly empty environment around *C211*. Now, blind motion attempts soon create a 'shockwave' of R, DR, and Ci that impedes forward progress. Current work is developing dynamic control methods to alter the protocell aspect ratio, and we expect more advanced Ci will also help clear the way.

Yes, there is much to do, but also look how far we have already come. We have a growing palette of spatial and distributed programming patterns (topological invariant maintenance, gradient formation, leader election, etc.)—all familiar in other contexts, but now united under an *indefinitely scalable* architecture. And the goal of *robust multicellular programmability* for that architecture is closer than ever.

Indefinitely scalable soft alife is coming.

Acknowledgments Elena S. Ackley implemented the **ulam** compiler underlying SPLAT. Dan Cannon demonstrated a 'cloud around a protocell' mechanism before the current simulator even existed, but this author didn't fully appreciate its power. *T2 Tile Project* subscribers provided valuable focus during the writing of the paper, and anonymous reviewer comments improved its quality.

References

Abelson, H., Allen, D., Coore, D., Hanson, C., Homsy, G., Knight, Jr., T. F., Nagpal, R., Rauch, E., Sussman, G. J., and Weiss, R. (2000). Amorphous computing. *Commun. ACM*, 43(5):74–82.

Ackley, D. H. (2013). Bespoke physics for living technology. *Artificial Life*, 19(3_4):347–364.

Ackley, D. H. (2016). Indefinite scalability for living computation. In *Proc. of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 4142–4146.

Ackley, D. H. (2018). Digital protocells with dynamic size, position, and topology. *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 83–90.

Ackley, D. H. and Ackley, E. S. (2015). Artificial life programming in the robust-first attractor. In *Proc. of the European Conference on Artificial Life (ECAL)*, York, United Kingdom.

Ackley, D. H. and Ackley, E. S. (2016). The *ulam* programming language for artificial life. *Artificial Life*, 22(4):431–450. https://dx.doi.org/10.1162/ARTL_a_00212.

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

Ackley, D. H., Cannon, D. C., and Williams, L. R. (2013). A movable architecture for robust spatial computing. *The Computer Journal*, 56(12):1450–1468. http://dx.doi.org/10.1093/comjnl/bxs129.

Banzhaf, W. and Yamamoto, L. (2015). *Artificial Chemistries*. MIT Press.

Cappello, F., Geist, A., Gropp, B., Kal, L. V., Kramer, B., and Snir, M. (2009). Toward exascale resilience. *IJHPCA*, 23(4):374–388.

Clement, A., Wong, E., Alvisi, L., Dahlin, M., and Marchetti, M. (2009). Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 153–168, Berkeley, CA, USA. USENIX Association.

Dijkstra, E. W. (1974). Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644.

Dolev, S. (2000). *Self-stabilization*. MIT Press, Cambridge, MA, USA.

Fontana, W. (1992). Algorithmic chemistry. In Langton, C. G., Taylor, C., Farmer, J. D., and Rasmussen, S., editors, *Artificial Life II*, pages 159–210, Redwood City, CA. Addison-Wesley.

Meyer, T. and Tschudin, C. (2009). Chemical networking protocols. In *Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*.

Orhai, M. and Black, A. P. (2012). Approximate parallel sorting on a spatial computer. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, RACES '12, pages 61–66, New York, NY, USA. ACM.

Saltzer, J. H., Reed, D. P., and Clark, D. D. (1984). End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288.