The *ulam* Programming Language for Artificial Life

David H. Ackley*,**
University of New Mexico

Elena S. Ackley†
Ackleyshack LLC

Keywords
Robust first computing, best effort computing, movable feast machine, asynchronous cellular automata

Abstract Traditional digital computing demands perfectly reliable memory and processing, so programs can build structures once then use them forever—but such deterministic execution is becoming ever more costly in large-scale systems. By contrast, living systems, viewed as computations, naturally tolerate fallible hardware by repairing and rebuilding structures even while in use—and suggest ways to compute using massive amounts of unreliable, merely *best-effort* hardware. However, we currently know little about programming without deterministic execution, in architectures where traditional models of computation—and deterministic ALife models such as the Game of Life—need not apply. This expanded article presents ***ulam***, a language designed to balance concurrency and programmability upon best-effort hardware, using lifelike strategies to achieve robust and scalable computations. The article reviews challenges for traditional architecture, introduces the *active-media* computational model for which ***ulam*** is designed, and then presents the language itself, touching on its nomenclature and surface appearance as well as some broader aspects of robust software engineering. Several ***ulam*** examples are presented; then the article concludes with a brief consideration of the couplings between a computational model and its physical implementation.

1 A Language for Soft Strong Artificial Life

We present ***ulam***, a new programming language for the research and development of indefinitely scalable *soft strong artificial life*—software creatures, embodied within some sort of digital machinery, that produce benefits for society. The language—mostly named after Stanislaw Ulam for his pioneering contributions especially in cellular automata [37]—is designed to work on *best-effort computing* hardware [4], where deterministic execution is not guaranteed. As a result, ***ulam*** programs—including the examples shown herein—are typically organized less like traditional algorithms than like living processes, employing mechanisms like reproduction, swarming, and growth and healing.

Why does the world need a new programming language, when it already has so many? Although ***ulam*** presumes an unreliable, distributed computational model, one could certainly imagine implementing such a model using, say, a Python script duplicated across a networked array of Raspberry Pi computers—no new language needed. Indeed, if the main goal were to get output from some particular computation, such a direct approach could make a lot of sense. However, as with most programming languages, ***ulam***'s primary purpose is less to implement any given task than to provide

* Contact author.
** Department of Computer Science, University of New Mexico, Albuquerque, NM 87131, USA. E-mail: ackley@cs.unm.edu
† Ackleyshack LLC, P.O. Box 993, Placitas, NM 87043, USA. E-mail: esa@ackleyshack.com

Artificial Life 22: 431–450 (2016) doi:10.1162/ARTL_a_00212

a framework for envisioning, expressing, and reasoning about—*as well as* executing—a particular style of physically realizable dynamical behavior. ***ulam*** provides significant gains in conciseness and expressiveness for its targeted best-effort computations, and that has enabled us to achieve greater levels of composability and complexity than we dared to attempt with our prior C++ and Java implementations.

Bedau [12] distinguished *hard*, *soft*, and *wet* varieties of artificial life research as those that employ hardware such as robots, or just software running inside computers, or actual chemical preparations and reactions. There are also varied *weak* and *strong* philosophical positions (e.g., [35]) about how best to understand the purpose and results of artificial life work. The goal of weak ALife we take to be living system *models*—abstract conceptual entities assessed primarily by their scientific, pedagogical, or artistic values. We take the goal of strong ALife, by contrast, to be living system *instances*—embodied physical entities assessed primarily by the value of the work they perform—judged as living, ultimately, by their ability to *make* a living.

This article is an expansion of [5], with updates throughout and primary new material including a fuller treatment of the ***ulam*** language; a new section connecting the ***ulam*** design to robust software engineering; and a new example and discussion about an unexpected consequence of indefinite scalability. Section 2 motivates computer architecture based on living systems design, with programming used to shape the dynamics of *active-media* systems—unbounded but unreliable systems where energy costs are prepaid to encourage parallelism. Section 3 presents a few *best-effort slogans* both as antidotes to determinism and as prototype design patterns for living computation. Then Section 4 introduces ***ulam*** in the context of robust software engineering, and Section 5 presents several examples. Finally, Section 6 stresses the links between the computational and the physical, and Section 7 offers some concluding thoughts.

2 Beyond Determinism

As the hegemony of CPU and RAM declines, for the first time in decades significantly new computer architectures are appearing, such as the nothing-but-net neural architecture of IBM's TrueNorth [29]. With the potential on the horizon for a major evolutionary transition in computer architecture, it is an opportune time to reconnect with first principles before shortlisting successors. The result of such a process, we believe, will be the recognition of artificial life as a (perhaps *the*) major force driving future architectural innovation.

2.1 Escape from the SDA

Serial deterministic computing based on CPU and RAM is a vast attractor, a valley deep and wide, in a notional space of all possible models of computation, including parallel, probabilistic, and analogue methods. This *serial deterministic attractor* (SDA) is laced with interlocking design decisions—such as binary number representations and the general exaltation of software efficiency—surrounding its core demand for logical correctness. The inherent fragility of extremely efficient software can be masked by extremely reliable hardware [3]—but only until a fault, or a bug, or an attacker, appears.

Although the SDA's robustness and security properties are dubious, and its scalability is rapidly dwindling [19], the assumption of deterministic execution has been so dominant for so long that alternatives may seem unthinkable. One might imagine that fields like fault tolerance (e.g., [25]) or probabilistic algorithms [26] fall outside the SDA, but by "virtually guaranteeing" deterministic execution, they actually entrench it. The same is true of many other nontraditional but still deterministic models, such as synchronous cellular automata (e.g., [38, 37, 36]), data-flow machines and systolic arrays (e.g., [16, 17]), and asynchronous circuit-level techniques such as GALS and RALA [27, 21].

Probabilistic cellular automata (PCAs) (e.g., [22, 11]) do go decisively beyond determinism, and they are general enough to embrace the kind of models we explore—but their motivations and methods are sharply divergent from the present effort. PCA work often presumes simple and stylized noise models, and proceeds—preferably by formal analysis—to derive insights into equilibrium

distributions and other system properties. But when such research begins by postulating a state transition matrix, the small matter of actual PCA programming is silently assumed away. Yes, the transition matrix is a powerfully general device; no, you don't want to program in it.

In addition, PCA models generally assume quite simple and low-order departures from determinism, such as i.i.d. update probabilities or random permutations—but as we shall see in Section 6, when moving beyond deterministic hardware to achieve indefinite scalability, the interactions between physical and computational levels can be sufficiently complex that the applicability of simple PCA models is far from certain.

Recently, there have been some serious programming research efforts that, while remaining mostly traditional, do explicitly abandon determinism and accept some small output errors—often with the motivation of increased parallel efficiency (e.g., [19, 20, 30, 32]). We cheer all such efforts, but worry they may fail to gain traction because their incremental practicality leaves them struggling up the sides of the SDA valley, with all the downhill directions behind them.

2.2 Colonize the RFA

There is at least one fundamental alternative to the SDA, which we here call the *robust-first attractor* (RFA), in the space of all possible models of computation. We have been breaking trail in the RFA for some time [6, 3, 7, 2, 8] and can report it is strikingly unlike the SDA, but at least as vast: It is a natural way to understand the computational properties of *living systems*, which have always made do without the luxury of deterministic execution.

Life fills space, as long as suitable resources are available; every RFA architecture must do the same, and that core demand for *indefinite scalability* is surrounded by interacting design decisions often deeply complementary to the SDA's. A von Neumann machine by itself simply isn't an RFA architecture; it is incomplete, and thus unevaluable, until a method is defined for tiling unbounded space with it ([2] has further discussion).

Most software-based artificial life models are designed to run on single von Neumann machines.[1] Unsurprisingly, therefore, the properties of such models typically depend critically on deterministic execution, as typified by the utter collapse of constructs in Conway's Game of Life when facing even mild asynchrony ([14]; see also [13]).

Determinism is a property of the small and the fragile; it is fundamentally misaligned with living systems. It warps our expectations; it is time to move on.

2.3 Programmable Active Media

SDA models are well suited to implementation in passive, "cold" materials, where uniformity rules, change is rare, and free energy is expensive—conditions where, indeed, living systems may survive but will rarely thrive. However, some environments are diverse in space, dynamic in time, and energetically rich, bountiful, like a rain forest or a sunny day at the shore. We abstract such circumstances into active-media computational models—unbounded spatial architectures in which each discretized location performs logical state transitions based on its local neighborhood, but with uncertain and variable frequencies and only limited reliability.

An active medium can change spontaneously and is inherently nondeterministic. In a *programmable* active medium we get to pick its state transition function—to specify, up to reliability limits, that certain neighborhood patterns shall stay constant (like memory, say) while others produce transitions like a processor or a data transport, or, indeed, act like different types of hardware at different moments. The state transition function we supply is executed asynchronously in parallel across the medium, avoiding overlapping state transitions, again, with good but not guaranteed reliability. It may become possible to implement programmable active media in, say, DNA (e.g., [18, 34]); it is already possible in electronics [7, 34].

1 Though there have certainly been exceptions, both proposed [31] and implemented [1]. Additionally, powerful modeling systems like Ready [23], though still fundamentally deterministic, now exploit many-core parallelism.

2.4 A New Deal for Hardware and Software

Clearly, compared to SDA architectures, the active-media model presents a very different division of labor, as traditional hardware components like "processor" and "memory" and "bus"—and their floorplanning—are placed largely under software control. This refactoring will presumably incur a hardware price–complexity penalty something like FPGA versus ASIC or worse—but that, in turn, may be more than offset by enabling new optimizations akin to RISC versus CISC, combined with the hair-down liberation of merely best-effort hardware determinism.

So, while the programmable active-media framework is likely a splendid deal for hardware, it may seem a brutal one–two punch for software, stunned by nondeterminism from below and then flattened by expanded mission responsibilities from above. But here's the thing: On the one hand, the software engineering job *should* be harder, because its relative simplicity was purchased with precisely those von Neumann machine features—a single processing locus, uniform passive memory, reliability all on hardware—that led to its Achilles' heels of unscalability and unsecurability. Serial determinism was a simple, sensible starting point, but software engineering and many related fields have emerged since von Neumann's time, and we now know quite a bit about constructing, managing, and evolving complex systems. From the RFA looking back, for software still to be demanding general pointers and flat RAM and cache-coherent global determinism seems like clutching a blankie. The future will arrive anyway.

That said, and on the other hand, software's big promotion becomes less terrifying as we get down to work, because, like hope from Pandora's box, "best effort" wafts upwards from the nondeterministic hardware into the software as well. As a system component, we'll do our best with what we've got and what we get, but if things go really wrong, we can simply delete ourselves and let our kin cover for us. Correctness and robustness are measured by degrees and circumstances in living systems; in the RFA they are highly respected qualities rather than merely purported necessities.

3 Principles of Active-Media Programming

To help make the RFA more concrete and clearly distinct from the SDA, in this section we offer some RFA slogans or design principles, with brief ALife motivations or implications. The end of Section 2.4 above concerns a principle that might be called *There's always dying*; here are five more:

- *Happy and you know it:* Make the goals obvious.
- *Space is the place:* Space is the core data structure.
- *Embrace the race:* Just help the better answer win.
- *Chance it:* Replace state with statistics.
- *Use it or lose it:* A cycle saved is a cycle wasted.

Happy and you know it: The key to robustness is effective redundancy, and one excellent approach is to give subcomponents not just tasks to do but also ways to measure their own success. Unit tests are a simple SDA example; in the RFA, geometric goals ("Bigger should be lower") and local consistency rules ("If I'm linked to you, you should be linked to me") allow the execution of external productive computations to be combined with ongoing internal processes like machine construction and maintenance.

Space is the place: The SDA tries to obliterate space using random-access memory, then acts all surprised by buffer overflows; the RFA uses space as the backbone organizing principle for both access control and computation, like the cat that's picky about who's allowed near, but then grooms everything in reach. The ALife community is largely in good shape here: A great strength of typical ALife models is their fundamentally spatial organization, unlike, say, typical genetic algorithms.

Embrace the race: The SDA generally abhors race conditions,[2] but in the RFA, with many components interacting and making things better locally, a race can be, not a regrettable shame to be hidden, but a fine tool for making a larger-scale decision, which may then be amplified by spectators for subsequent processing. ALife models also do well with this—often identifying their race conditions with names involving "selection."

Chance it: The SDA is typically deterministic even when it's not supposed to matter, as in breaking ties or sizing a supposedly ample buffer or time delay. But with determinism off the table, many RFA tasks can be dramatically simplified by replacing state with probabilities, and rather complex dynamics can be implemented using remarkably little per-object state, as demonstrated in Section 5.3 below. Alife models are mixed on this, but the field faces some hard unlearning.

Use it or lose it: The active-media abstraction is a programmable space in which energy is a nonrefundable sunk cost, and even though actual power is limited, this idealization shifts the discussion from the stultifying task of minimizing the cost of energy consumed to the galvanizing task of maximizing the value of energy provided. Parallel computing involves many values changing frequently; other things being equal, the RFA programmer seeks to minimize the *average change age per site* in some productive way.

4 The Ulam Programming Language

Such slogans help to establish goals and shared context but are no substitute for executable code. For several years, we have been exploring a tile-based indefinitely scalable architecture called the *movable feast machine* (MFM) [6, 7, 2]. Over the last two years we have been developing an open-source C++ MFM engine [9], which is currently deployed in `mfms`, a Linux-based thread-per-tile simulator running on conventional multicores, but is designed and written for eventual cross-compilation onto indefinitely scalable tile-per-tile physical hardware.

We have also been developing an open-source compiler for a new language we call ***ulam*** [10] that generates code to run on the MFM. We give only a few bullet points for flavor to get started here—there's runnable code in the next section—but ***ulam*** deliberately looks and works much like a conventional object-oriented procedural language:

- The command-line driver, `ulam`, is a Perl script. The ***ulam*** compiler itself, `culam`, is written in C++, and its output is heavily templated C++, so that despite weird data sizes and packing (see below), the `g++` compiler downstream can sometimes find quite delicious assembly code sequences. The key compilation output is a `.so` shared library, which can be dynamically linked into the `mfms` simulator from the command line, or packed into a cryptographically signed runnable container (`.mfz` file, "movable-feast zip") along with sources, simulator starting configurations, and arbitrary files.
- Groups of 1 to 32 bits can be declared as primitive values, using a variety of semantics (see Table 1). Note that ***ulam*** is quite strict about what operators apply to what types, largely to support language robustness features (see Section 4.5). One-dimensional arrays of zero or more items are provided, bit-packed up to 32 total bits.
- The ***ulam*** analogue to an object instance is called an `Atom`, but an `Atom` is also analogous to a word of memory in a tagged architecture, so *every object is the same size*—96 bits in the current design, with 25 reserved for the object type and error correction. The analogue to a class is called an `element`, which may define constants, parameters, nonstatic methods, and up to (96 − 25 =) 71 bits of data members.

2 And even those rare authors who accept nondeterminism usually see races as no more than tolerable [30, 15], although there are exceptions (e.g., [28]).

Table 1. ***ulam*** primitive types, available in all bit widths from $k = 1$ to the default of 32, except that `Bool` defaults to 1 bit, and rounds down to odd widths to avoid ties.

ulam type	Interpretation	Operators
`Unary`(*k*)	Base 1 (population count)	`- + * / == != < > <= >=`
`Unsigned`(*k*)	Base 2	`- + * / == != < > <= >=`
`Int`(*k*)	Two's complement	`- + * / == != < > <= >=`
`Bool`(*k*)	Boolean (unary majority)	`! == != && ||`
`Bits`(*k*)	Uninterpreted bit values	`== != << >> & | ^`

- There is composition but at present no inheritance; the analogue to a struct is called a `quark`, which may be any size from 0 to 32 bits, and which may be templated with compile-time numeric constants. There is also `union` with the typical meaning. An `as` keyword introduces a *conditional declaration* akin to a Java "instanceof" plus a cast, to access specific class members inside `Atom` instances.

Overall, an ***ulam*** program consists of a compiled set of class definitions, plus a finite initial layout of `Atoms`. There is no `main()`, and by default the initial layout is empty, so in the simplest case no computation occurs at all, unless and until some external influence causes one or more `Atoms` to be created.

4.1 Language Design Decisions for Robust Software Engineering

The term "robustness" can seem frustratingly vague, compared to the crystalline purity—at least in principle—of the term "correctness." Compared to most machines, living systems are famously robust—but that's when they're in environments where they evolved.

In the general case, there are just too many ways for systems to fail—due to latent defects, to unexpected conditions or interactions, to age or attack or accident. Nothing can survive forever, so—without referring to some actual device in some actual environment—how should failures be prioritized?

In lieu of a strong formal theory of robust programming, in the rest of this section we illustrate our qualitative approach to abstract robustness in the ***ulam*** language design, touching on its memory and processing model, and its data and operator semantics.

4.2 Reduce Deterministic Memory

Computer working memory has a tough job: It must reliably store values indefinitely, yet also reliably alter those values at a moment's notice. Unlike typical procedural languages, ***ulam*** provides no direct language support for persistent memory—there is a function-call stack, but that only lasts for a single event, and there is no dedicated heap or general-purpose main memory. "RAM" support, such as it is, is provided by the `EventWindow` quark in the standard library, which provides read/write access to a small, two-dimensional patch of *sites* (Figure 1), each of which can contain one atom, perhaps with a constant amount of additional state (for example, associated with I/O in the "third dimension"). Event window accessing is relative to the atom having the event, which is, by definition, located in the center site at (0,0). During a single event, no assurances are given about the contents (or even the existence) of any sites outside the event window—forcing the ***ulam*** element programmer to stop hoarding data and make hard choices up front about what to represent explicitly within the atom, and what to build up stigmergically instead.

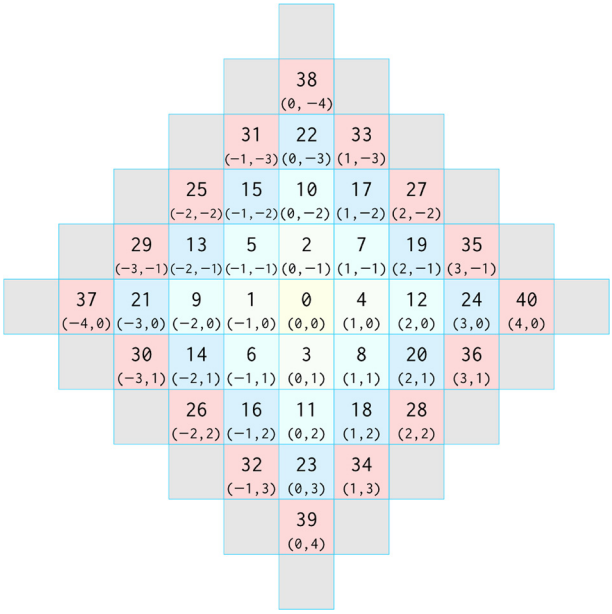


Figure 1. The neighborhood for a state transition, provided by the `EventWindow` quark in the standard library, comprises the 41 sites within Manhattan distance 4 of the event center at site index 0.

4.3 Reduce Deterministic Processing

The MFM provides best-effort deterministic computation for the duration of a single state transition that updates a single event window. Since there is no privileged `main()` method, from the point of view of an element, execution occurs during a call to a handful of special methods (see Table 2), especially the `behave()` method implementing that element's state transition function.

Note the MFM's commitment to best-effort single-event determinism—though weak compared to traditional serial determinism—is just one possible engineering tradeoff between programmability and performance for an indefinitely scalable system. With this approach, for example, while one given site has an event, its forty nearest neighbors cannot. There is also significant locking overhead to provide best-effort site consistency when an event window crosses tile boundaries ([7] has details).

In a more aggressive design, by contrast, overlapping event windows could be permitted, and their site accesses would race against each other in the memory system and intertile communications, much as poorly written thread-based code does in multicore systems today. As we have gained experience programming robust ***ulam*** elements, such a prospect is becoming less utterly terrifying than it was during the original movable-feast design—but we leave any serious consideration of such possibilities to future work.

Table 2. ***ulam*** privileged methods for elements (*E*) and/or quarks and unions (*Q*).

Special method	On	Purpose
`Void behave()`	*E*	Perform event
`Int test()`	*EQ*	Run unit tests
`Int toInt()`	*Q*	Custom cast to `Int`
`T aref(Int)`	*EQ*	Custom array read
`Void aset(Int, T)`	*EQ*	Custom array write

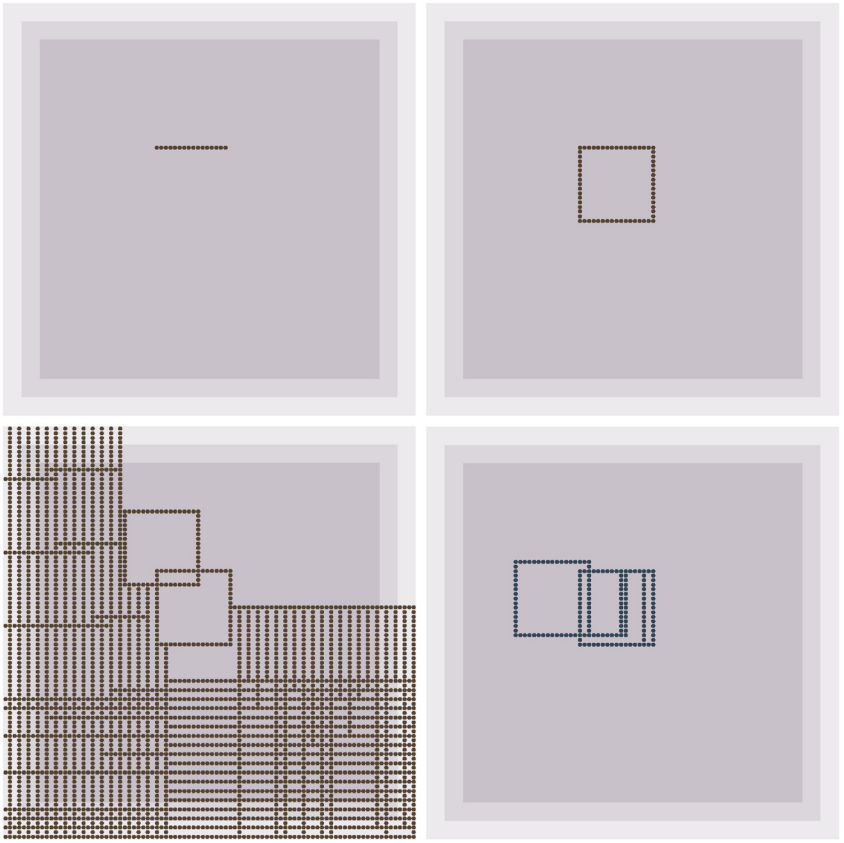


Figure 2. Four separate simulations. *Top left:* Sixteen-atom self-constructed `Line` segment. *Top right:* Self-constructed `Box`. *Bottom left:* Example of catastrophic `Box` failure mode. *Bottom right:* Example of bounded `Box2` failure mode. See text.

4.4 Reduce Primitive Sizes

One blindingly obvious robustness principle took us embarrassingly long to see: *Nonexistent bits cannot fail*. As discussed above, we designed ***ulam*** to support variable-width primitives because the `Atom` is so small that down-to-the-bit space management is often inevitable—but eliminating unneeded bits also helps mitigate bit failures. A single bit flip in an `Unsigned(32)` could change the stored value by billions; if the programmer's actual intent was to store an integer percentage, then, say, an `Unsigned(7)` wins in space efficiency *and* robustness.

A fundamental robustness goal is to avoid creating leverage the program doesn't need—and excess high-order bits are just one example. As a concrete example, our experience developing a self-constructing `Box` element illustrates how easy-to-overlook software engineering concerns can have significant robustness consequences.

The `Box` element exploits the fact that the `EventWindow` quark supports neighborhood access under any of the eight square symmetries—expressed in code by type `Symmetry`, which is a typedef for `Unsigned(3)`. Starting from a `Line` element that constructs a horizontal line segment (Figure 2, top left), we made a reusable abstraction called a `QLine`, and created a `Box` element[3] composed of a `QLine` plus an additional symmetry data member, indicating which way this particular line segment should grow. When one side of the box was complete, it would copy itself to seed another line

3 See the Appendix for commented ***ulam*** source code of `Line` (and its underlying quark `QLine`) as well as `Box2`, which, aside from naming, differs from `Box` only at line 12 of Appendix A.2.

segment if there was an empty site beyond—but with the *next* rotational symmetry in the copy's data member. After four such line segments, the `Box` would be complete (Figure 2, top right).

The use of `QLine` inside `Box` shows one way that composability can work in ***ulam***, providing reasonably clean modular code and good separation of concerns between creating a line segment on the one hand, and placing and orienting it on the other, to create a more complex structure. For typical programming languages the example could easily end there, but here we are also interested in the robustness of the solution, and to that end the simulator provides a variety of mechanisms to damage computations deliberately, either via ongoing automatic processes or manually operated tools.

While investigating the dynamics of `Box` using the xray simulator tool to induce random atomic bit flips, we were surprised by the virulent, nearly spacing-filling failure modes that sometimes occurred (Figure 2, bottom left.) Debugging revealed that a bit flip in the symmetry data member was selecting the *flipped* `EventWindow` symmetries, causing the next rotation to turn the wrong way, so the confused `Box` spreads rather than closing. That led us to write element `Box2`, which stores only the two bits needed for the rotational symmetries, and we found that while xray probing triggered construction of misaligned `Box2` atoms, no runaway metastases were observed (Figure 2, bottom right).

The robustness lesson is: Bits that aren't stored cannot be corrupted in storage. By using just two bits, to represent only the four rotations of the box sides, and then expanding those bits to the full three-bit `EventWindow.Symmetry` as needed, the third symmetry bit is always a freshly generated zero, rather than a copy of an ever-aging bit from memory. Though in principle that new zero could also fail, during its brief existence the damage would be limited because the `Box` "germline" remains unaffected.

Beyond that, a software design lesson may also be gleaned here about specificity versus generality. It is inelegant for `Box2` to define a custom symmetry type directly in terms of primitives (Appendix A.2, line 12). It would be cleaner to use a type provided by `EventWindow` like `Box` did (as suggested by Appendix A.2, line 11). Without disrupting backward compatibility, a future `EventWindow` redesign may recognize that the existing general `EventWindow.Symmetry` can be decomposed into a more specific two-bit rotation and a one-bit flip, and expose those types separately as well, allowing simultaneously improved space efficiency and robustness without violating modularity.

4.5 Offer Robust Semantics

ulam is a strongly typed language, which we view as a robustness and efficiency feature both; and for parsing convenience as well as clarity, type names are distinguished lexically by an initial uppercase letter. Here we touch on some aspects of the ***ulam*** type system that involve robustness.

A major decision in the design of ***ulam*** is that arithmetic operations and casts *saturate* in the destination type. So after `a = 5;` and `a += a;`, for example, then `a` will be 7—if `a` was declared as `Unsigned(3)`.

Traditionally, arithmetic operations simply overflow when the result bit width exceeds the storage width. That's cheap to implement, and—according to the traditional thinking—if the answer is incorrect anyway, who cares *how* incorrect it is? But in best-effort computing, even though saturation and overflow both yield incorrect answers, saturation is often *less wrong*, and that matters for robustness.

Given that numeric types saturate, bitwise operations like shifts are defined to produce type `Bits`—an ordered collection with no aggregate numeric interpretation. If programmers really want to shift an `Int` to the left, say, and thereby possibly change the sign of the result, they must explicitly cast the `Bits` result back to an `Int`, and accept the consequences.

As a more obvious robustness feature, `Unary`, the overlooked black sheep of number formats, is a first-class type in ***ulam***. Though unary is only suitable for small values, small values are surprisingly common in programs—and for such values, unary numbers are as inherently robust as binary numbers are pointlessly efficient. The bit flip that can cause a huge error in binary causes an error of one in unary. Why take the risk if you don't have to?

Closely related to `Unary` is the ***ulam*** `Bool` type, which uses only odd bit widths and has value `true` if and only if its storage contains more ones than zeros. It is symmetric with respect to bit flips, unlike the traditional nonzero-is-true "bool," in which an accidental `false→true` transition is much more likely than the reverse.

For all of these design decisions, there is no guarantee they will help in any particular case. But, again, in best-effort computing, providing a guarantee is not required, and—more to the point—the *inability* to provide a guarantee is no excuse to give up and collapse into chaos.

5 Small *ulam* Examples

Here we present several examples of ***ulam*** code that we have created recently, embodying living principles like reproduction (uncontrolled and controlled) and group formation and action.

5.1 Fork Bomb

In Unix-derived operating systems [24], the *fork* system call initiates a new process, copied from the one that issued the *fork* call. Known colloquially as a *fork bomb*, a trivial program built around a loop like `while (true) { fork(); }` will rapidly swamp the machine with processes doing nothing but attempting to reproduce.

Figure 3 presents complete ***ulam*** code for an analogous runaway-reproducing fork bomb. Line 1 declares the language version in use. Line 7 declares an instance of the `Event-Window` quark as a data member. `EventWindow`, which conveniently consumes only zero bits of atomic state, offers the expected 2D neighborhood access methods, as well as a one-dimensional mechanism that often suffices, as here: The `self`—the atom having the event—is by definition at `ew[0]`, with increasing 1D array indices spreading outwards in 2D breadth-first as shown in Figure 1.

With up as "north," `ew[1]` is the adjacent site west, so one might expect `ForkBomb` to produce a westward-growing line. But the element metadata (inside the structured comment, at line 4) declares that all eight square rotations and reflections are valid for `ForkBomb`, so a random transform is chosen for each of its events. It quickly explodes into a ragged mass (Figure 4) that will eventually fill space, as each randomly sited event—and randomly chosen symmetry for that event—does or does not happen to convert a remaining empty site into another `ForkBomb` atom.

Running ***ulam*** on `ForkBomb.ulam` compiles it into C++ intermediate code, which is then processed by the standard GCC tools, yielding a custom ***ulam*** element library (`libcue.so`) that can be dynamically loaded into the MFM simulator—and, hopefully soon, into actual hardware tiles. Loaded into the simulator, the library adds the `FB` element to the available element palette, allowing computations like Figure 4 to be run immediately.

```
1  ulam 1;
2  /** Fork bomb.
3    \symbol FB \color #f00
4    \symmetries all
5  */
6  element ForkBomb {
7    EventWindow ew;
8    Void behave() {
9      ew[1] = ew[0]; // Make more me!
10   }
11 }
```

Figure 3. A complete ***ulam*** element. Copies itself from the event window center (`ew[0]`) to `ew[1]`, which in this case might be any adjacent site. See text.

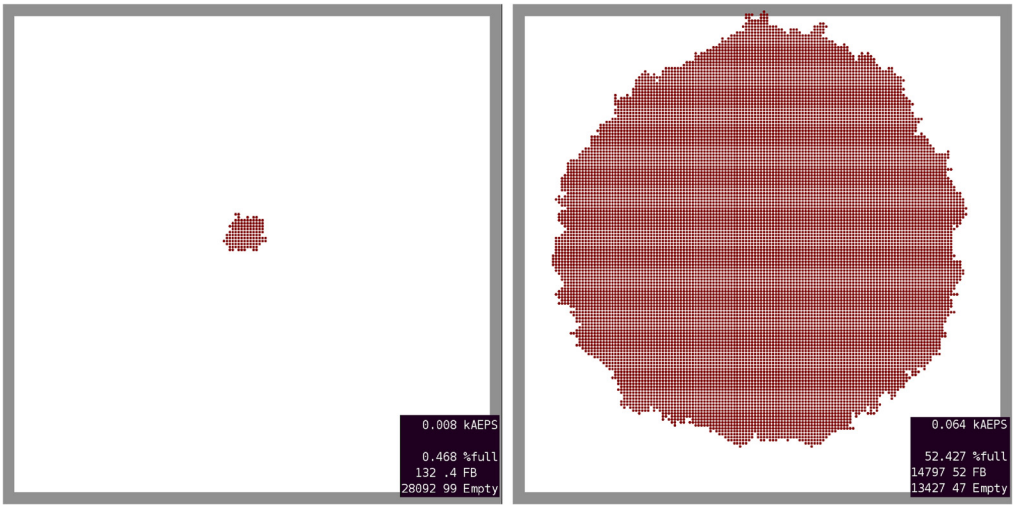


Figure 4. Uncontrolled (and uncontested) growth of a single initial `ForkBomb`. *Left:* After an average of 8 events per site (8 AEPS). *Right:* After 64 AEPS.

5.2 Telomere

The forkbomb is so simple and obvious that it's sort of a "Hello world!" for active media, but its cancerous rampage is a poor example of ALife programming. Inspired by the *telomere* DNA sequences that shrink during reproduction, Figure 5 illustrates a reusable ***ulam*** software component that provides controlled reproduction of a single starting atom into a clonal population of $2(2^{\text{width}} - 1)$ atoms, assuming all clones survive and spread out sufficiently. The group-forming "mob" in Section 5.4 below uses this strategy.

5.3 Stochastic Timer

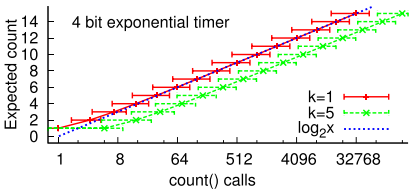
In embedded systems, a *watchdog timer* is often used as a last-ditch backstop to monitor critical processes that *should* complete in a timely fashion, but *might* fail to do so—for any reason, including a

```
ulam 1;
quark Telomere(Unsigned width) {
  typedef Unsigned(width) Tail;
  typedef EventWindow.SiteNum SiteNum;
  EventWindow ew;
  Tail age;
  /** Duplicate into ew[to] if self isn't
    too old and ew[to] is a live, empty site. */
  Bool dup(SiteNum to) {
    if (age < Tail.maxof) {
      if (!ew.isLive(to) ||
          !(ew[to] is Empty))
        return false;
      ++age; // Increment before copying!
      ew[to] = self;
    }
    return true;
  }
}
```

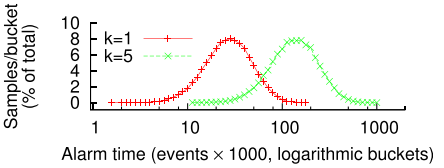
Figure 5. The `Telomere` quark `dup()` method offers controlled growth to elements containing it.

```
ulam 1;
/** A stochastic exponential timer; a template taking
  an exponential factor (exp) and a multiplicative
  factor (k). After a reset(), count() can be called
  approximately k*2**exp times before alarm() will
  begin returning true.
*/
quark Timexp(Unsigned exp, Unsigned k) {
  Random r; // PRNG is infrastructure, costs 0 bits
  Unsigned(exp) t; // costs exp bits
  Void reset() { t = 0; }
  Unsigned count() {
    if (!alarm() &&
        r.oneIn((Unsigned) (k<<t)))
      ++t; // Each tick about doubles in length
    return t;
  }
  Bool alarm() { return t == t.maxof; }
}
```

(a) Complete ***ulam*** code of a quark template.



(b) Timexp (4,1) counts powers of two.



(c) The alarm time distributions of Timexp (4, *k*) are symmetric on the logarithmic scale.

Figure 6. A stochastic timer. A four-bit `Timexp(4,1)` separates time scales over four orders of magnitude, combining fine-grained initial resolution with overall long duration.

software bug or a hardware fault. A watchdog timer's timeout value is usually set very generously to avoid interfering with normal operations. In ***ulam***, using a normal binary number big enough to count to thousands or millions of events would consume ten or twenty bits of the scarce atomic bit budget—which seems especially profligate in that the exact moment of a watchdog timeout is all but irrelevant anyway.

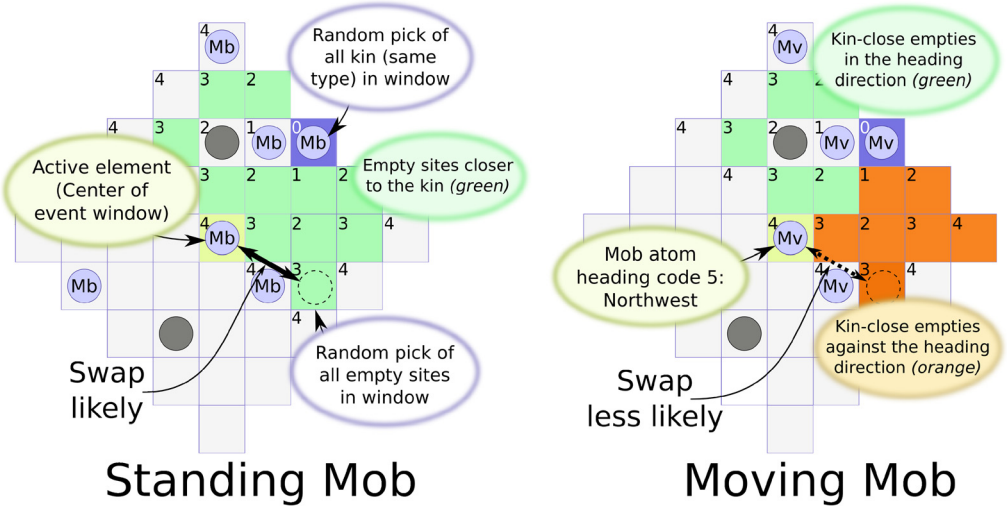


Figure 7. Two mob atom types evaluate their options, having picked at random from the empty sites (green or light gray) and a random other mob atom. *Left:* The basic mob rule: Swap if the empty is closer to the kin. *Right:* Mobs induce drift by discouraging moves against a heading direction (orange or dark gray). See text and Figure 8.

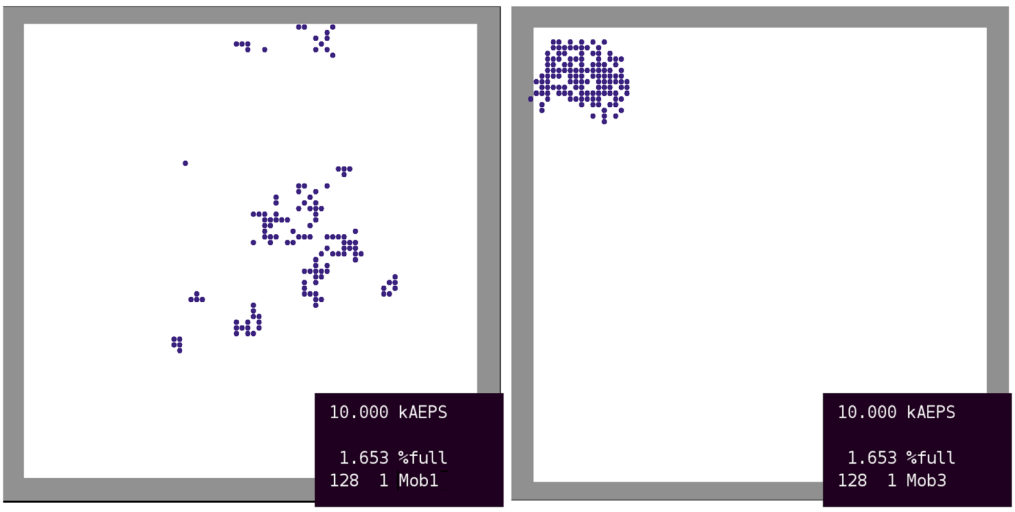


Figure 8. *(Left)* At 10 kAEPS a mob using just the basic mob rule (Figure 7, *left*) has broken up, but *(right)* adding statistical gravity and a directional bias (Figure 7, *right*) yields a long-lived, slowly moving mob. Mobs began as single centered atoms.

Figure 6a shows an alternate approach, saving most of those bits by using statistics instead of state, as mentioned with the "chance it" principle in Section 3. `Timexp` is a quark template taking a bit size parameter (`exp`), and a multiplier (`k`). Here `exp` specifies both its range of counting ability, and how much it consumes from the atomic bit budget; `k` scales the entire curve.

So a `Timexp(4,1)` quark, for example, can be dropped into any element with four bits to spare, and its `count()` method can be called thousands of times, with very high probability (Figure 6b), before its `alarm()` method returns `true`. When its `t` data member is 0, `k<<t` is 1 and so `oneIn()` returns `true` for sure, but each successive increment doubles the odds and on average takes twice as long.

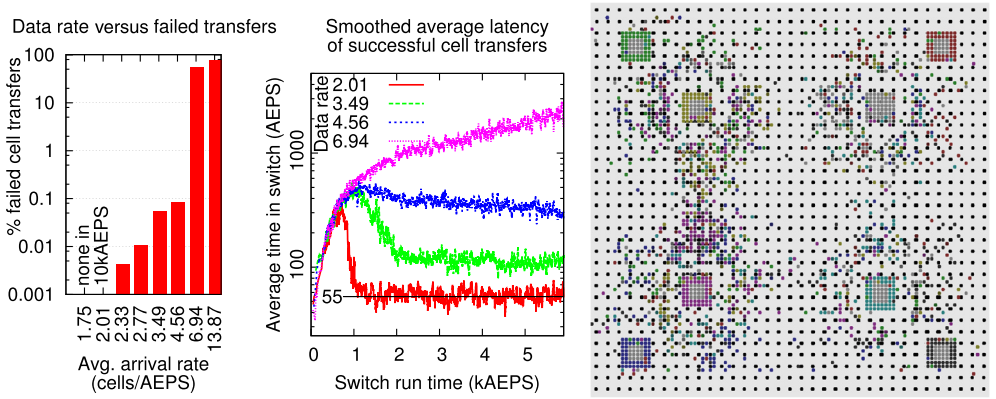
5.4 Mob Rule

This fourth example demonstrates both cohesion and mobility in a single structure—a *mob*—that's larger than an event window but smaller than the whole universe. A `Telomere(3)` quark induces a single mob atom to form a cluster of 128 clones, all following the *mob rule* (Figure 7). That proved insufficient for mob cohesion (Figure 8, left), but it works if we skip some mob moves with a probability that grows with the number of mob atoms in our event window, a fix we call *statistical gravity* (Figure 8, right).

Classical swarms such as the celebrated *boids* model [33] employ real-valued positions and velocities, perform deterministic updates, and rely on long-distance interactions when the swarm density is low. A mob can be seen as a simple kind of swarm tailored for the MFM computational environment, making stochastic, discrete movements based on only the limited information available in a single event window (Figure 1). Mobs are new, but we hope they find use—as either code or inspiration—as a medium-scale mobile spatial data structure. They are slow and sloppy; but note that all that dynamics—the managed growth, the group cohesion, and the controllable, directed movement—costs just one byte per atom.

5.5 Data Switch

As a final example, to explore and stress ***ulam***'s programmability, we developed a toy data switch, designed to carry fixed-size data cells between eight bidirectional ports. We made `Router` atoms that self-assemble into a grid, while gossiping amongst themselves to derive spatial gradients towards eight `Port` atom clusters, which emit and consume gradient-following data `Cell` atoms, each carrying a four-byte payload destined for a random other `Port`.



(a) Cell arrival rate versus % lost.

(b) Data switch latency versus machine uptime.

(c) Snapshot at 10 kAEPS; cell arrival rate = 4.56.

Figure 9. Performance of the 8-port switch: The data loss (a) is low until a critical arrival rate is reached; the initial average latency (b) is high until the routing grid is constructed. Colored square borders in (c) are `Port` atoms, some in the color of just-received data; scattered dots are in-flight data `Cells`; the grid of `Router` atoms can be seen against the gray background. See text.

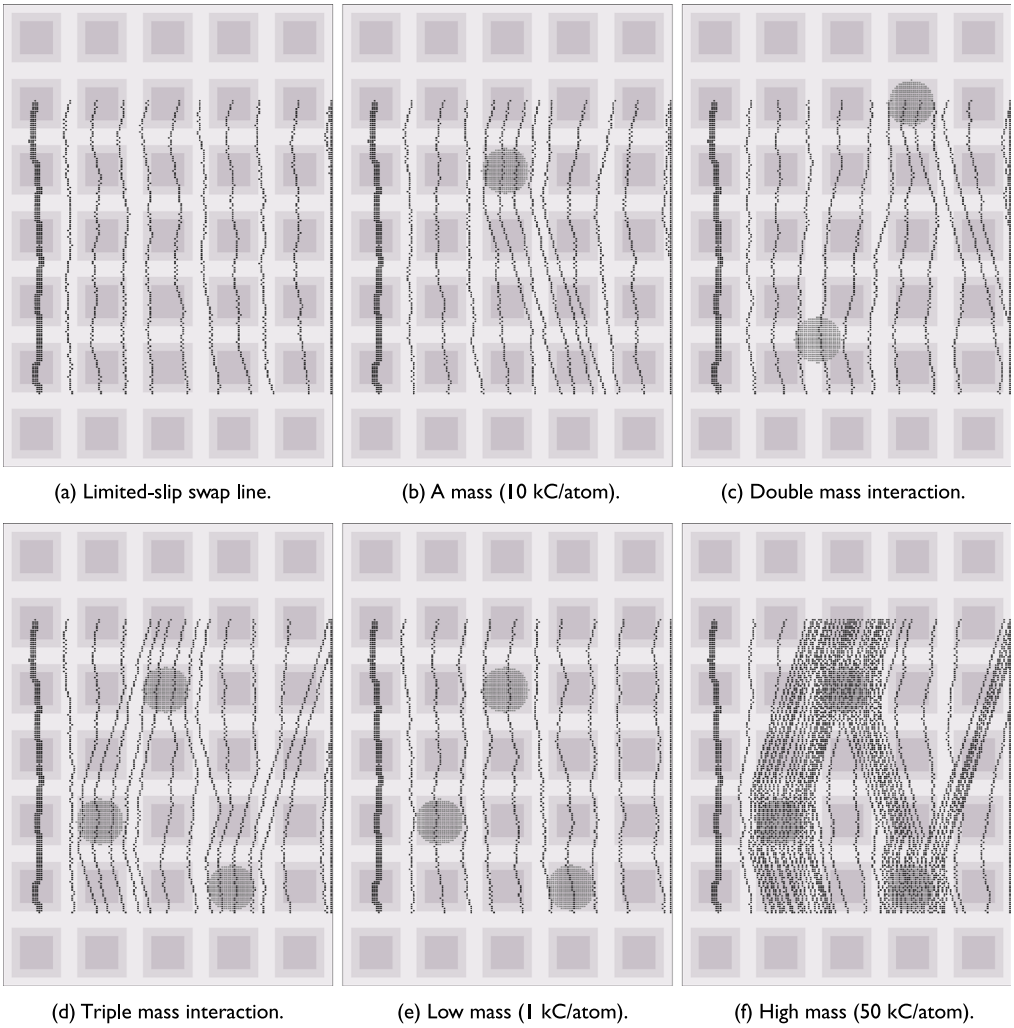
In this demo, about 550 lines of switch-specific ***ulam*** code compile into about 30K lines of very stylized C++; Figure 9 shows some data (9a, 9b) and a day-in-the-life image redrawn from a screenshot (9c). This little switch knows nothing of packet reassembly and sometimes drops cells, and it's slow, jittery, prone to catastrophic crowding at higher data rates, totally impractical, and completely glorious.

6 Hardware Determinism versus the Real

Hardware determinism is a tremendously powerful and seductive assumption, providing—as long as it holds—*perfect isolation* of the computational level from the physical. We can be utterly ignorant of physics and energy and noisy reality as long as we can master programming, which grants us unquestioned immediate top-down control of everything our machine can do. It is absolute dominion, the dream of empire made real, and neatly wrapped up in a box. Reluctance to part with it is entirely understandable.

```
1  /** Limited slip swap line. \symbol SL \color #222 */
2  element SwapLine {
3    EventWindow ew;
4    Void behave() {
5      for (EventWindow.SiteNum slot = 1; slot < 41; ++slot) {
6        C2D c = ew.getCoord(slot); // Get site coord
7        if (c.getX() >= 0) continue; // Only look west
8        Atom a = ew[slot];
9        if (a is SwapLine) { // Found a straggler?
10         if (c.getY() == 0) { Empty e; ew[0] = e; } // On our line?
11         return; // Dead or waiting
12       }
13     }
14     ew.swap(0,4); // All caught up: Head east
15   }
16 }
```

Figure 10. An eastbound limited-slip swap line. See text.



(a) Limited-slip swap line. (b) A mass (10 kC/atom). (c) Double mass interaction.

(d) Triple mass interaction. (e) Low mass (1 kC/atom). (f) High mass (50 kC/atom).

Figure 11. Stroboscopic views of an eastbound limited-slip swap line. See text.

It's just, of course, that the box isn't growing as it used to, and the more boxes we try to lash together into one, the less immediate our control becomes, and the less perfect the isolation of levels, so determinism fails. For programmers that can be a frightening prospect indeed—but best-effort computing takes it for granted.

We are in only the earliest stages of exploring the risks and opportunities of interactions between the physical and computational levels. When a probabilistic cellular automaton presumes i.i.d. update probabilities as a model of physical variation, for example, that implicitly assumes that the impacts of failed level isolation move only from the physical to the computational. But of course the computational *is* physical; influences can move both ways.

Consider the `SwapLine` element in Figure 10. The portion of the loop at lines 8–12 checks if the event window contains any `SL` atoms to the west; if so, the current event ends (in suicide if the other atom is in the same row), but if we reach line 14, nobody's been left behind and the atom swaps itself one site east. Over time, a vertical line of `SL`s gradually heads east, keeping itself roughly vertical, as in Figure 11a, which shows the same line captured at 50-AEPS intervals, superimposed on a single grid of 5 × 7 simulated tiles.

But what is happening in Figure 11b? The circular clump of atoms in the middle of the third row seems to be holding back the swap line, but how? The `SL` code ignores everything except other `SL`s. The atoms in the clump do not *directly* affect the `SL`s in any way—but they are atoms of the element `Massive`, symbol `MA`, which does nothing but waste a configurable amount of time. A 10-kC `MA` atom, for example, iterates 10,000 times every time it has an event, and does nothing else. The encroaching `SL`s can swap the `MA`s with impunity—but only once they've been picked to have an event.

Since each tile is an independent processor (or thread, in the case of the simulator; the Figure 11 simulations were performed on a four-core machine), and since there is no attempt to synchronize them globally, nearly empty tiles simply perform more events per second than do tiles packed with `MA`. The `SL`s out in "free space," therefore, start to pull ahead, until the limited-slip mechanism begins to hold them back. As the "computational mass" of the `MA` is dialed lower (Figure 11e) or higher (Figure 11f), the impact on the swap-line progress is correspondingly modulated.

Details at the computational level affect the timings of events at the physical level, which in turn affect the states that arise back at the computational level. Influences flow both ways. In a metaphorical sense at least, computation time acts something like mass in a general-relativity framework. At the intertile spatial granularity, computationally heavy elements *slow* time around them.

There will be not just risks, but potentially great opportunities, in the interactions between the physical and computational levels. We shouldn't be hiding from those interactions; we should be exploring them.

7 Conclusion

The ***ulam*** compiler has been under heavy development since August 2014, and coincident with ECAL 2015, we announced the official version 1.0 release, complete with documentation, tutorials and Ubuntu packaging so that installation can be as simple as `apt-get install ulam` from a public personal package archive.

There is so much to be learned and relearned, designed and redesigned, implemented and reimplemented. It can seem positively daunting, but we hope to entice you or your students (or advisers!) to give it a try.

You might be a Ruby fan, or Python or Haskell or Java or Forth; some if not many of our language design decisions in ***ulam*** will almost surely not be your cup of tea; that's OK. We just need to keep our eyes on the prize, as stated at the outset: The goal is neither maximum parallel efficiency, nor maximum expressive purity. The goal is *indefinite scalability while balancing concurrency and programmability*—and to that end, robustness must be inherent not just in hardware, but across the computational stack.

We must explore and colonize and settle the RFA, but ultimately we will also leverage the technology that is now filling our society, after seventy years of relentless optimization in the serial deterministic attractor. There is nothing wrong with von Neumann machines that cannot be fixed by making them small and individually insignificant parts of an indefinitely scalable architecture.

Change is coming. ALife research can lead the way.

Acknowledgments

This work was supported in part by a Google Faculty Research Award, and in part by grant VSUNM201401 from VanDyke Software.

References

1. Ackley, D. H. (1996). ccr: A network of worlds for research. In C. Langton & K. Shimohara (Eds.), *Artificial life V. Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems* (pp. 116–123). Cambridge, MA: MIT Press.
2. Ackley, D. H. (2013). Bespoke physics for living technology. *Artificial Life*, *19*(3–4), 347–364.

3. Ackley, D. H. (2013). Beyond efficiency. *Communications of the ACM*, *56*(10), 38–40. Author preprint: http://nm8.us/1.

4. Ackley, D. H. (2016). Indefinite scalability for living computation. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (pp. 4142–4146). URL http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11987.

5. Ackley, D. H., & Ackley, E. S. (2015). Artificial life programming in the robust-first attractor. In *Proceedings of the European Conference on Artificial Life (ECAL) 2015* (pp. 554–561). York, UK. URL http://dx.doi.org/10.7551/978-0-262-33027-5-ch097.

6. Ackley, D. H., & Cannon, D. C. (2011). Pursue robust indefinite scalability. In *Proceedings of HotOS XIII*. Napa Valley, CA: USENIX Association.

7. Ackley, D. H., Cannon, D. C., & Williams, L. R. (2013). A movable architecture for robust spatial computing. *The Computer Journal*, *56*(12), 1450–1468. URL http://comjnl.oxfordjournals.org/content/56/12/1450.abstract.

8. Ackley, D. H., & Small, T. R. (2014). Indefinitely scalable computing = artificial life engineering. In *Proceedings of The Fourteenth International Conference on the Synthesis and Simulation of Living Systems (ALIFE 14) 2014* (pp. 606–613). Cambridge, MA: MIT Press. URL http://dx.doi.org/10.7551/978-0-262-32621-6-ch098.

9. Ackley, D. H., & Small, T. R. (2014). The MFM version 2 codebase. https://github.com/DaveAckley/MFMv2.

10. Ackley, E. S., & Ackley, D. H. (2014). The Ulam compiler for MFM programming language. https://github.com/elenasa/ULAM.

11. Agapie, A., Andreica, A., & Giuclea, M. (2014). Probabilistic cellular automata. *Journal of Computational Biology*, *21*(9), 699–708.

12. Bedau, M. A. (2003). Artificial life: Organization, adaptation and complexity from the bottom up. *Trends in Cognitive Sciences*, *7*(11), 505–512. URL http://www.sciencedirect.com/science/article/pii/S1364661303002626.

13. Beer, R. D. (2014). The cognitive domain of a glider in the game of life. *Artificial Life*, *20*(2), 183–206. URL http://dx.doi.org/10.1162/ARTL_a_00125.

14. Bersini, H., & Detours, V. (1994). Asynchrony induces stability in cellular automata based models. In *Artificial Life IV* (pp. 382–387).

15. Boehm, H.-J. (2012). Position paper: Nondeterminism is unavoidable, but data races are pure evil. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability, RACES '12* (pp. 9–14). New York: ACM. URL http://doi.acm.org/10.1145/2414729.2414732.

16. Borkar, S., Cohn, R., Cox, G., Gleason, S., Gross, T., Kung, H. T., Lam, M., Moore, B., Peterson, C., Pieper, J., Rankin, L., Tseng, P. S., Sutton, J., Urbanski, J., & Webb, J. (1988). iWarp: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88* (pp. 330–339).

17. Budzynowski, A., & Heiser, G. (2013). The von Neumann architecture is due for retirement. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems, HotOS'13* (p. 25). Berkeley, CA: USENIX Association. URL http://dl.acm.org/citation.cfm?id=2490483.2490508.

18. Canton, B., Labno, A., & Endy, D. (2008). Refinement and standardization of synthetic biological parts and devices. *Nature Biotechnology*, *26*(7), 787–793. URL http://dx.doi.org/10.1038/nbt1413.

19. Cappello, F., Geist, A., Gropp, B., Kal, L. V., Kramer, B., & Snir, M. (2009). Toward exascale resilience. *IJHPCA*, *23*(4), 374–388. URL http://dblp.uni-trier.de/db/journals/ijhpca/ijhpca23.html#CappelloGGKKS09.

20. Elliott, J., Hoemmen, M., & Mueller, F. (2014). Exploiting data representation for fault tolerance. In *Proceedings of the 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA '14* (pp. 9–16). Piscataway, NJ: IEEE Press. URL http://dx.doi.org/10.1109/ScalA.2014.5.

21. Gershenfeld, N., Dalrymple, D., Chen, K., Knaian, A., Green, F., Demaine, E. D., Greenwald, S., & Schmidt-Nielsen, P. (2010). Reconfigurable asynchronous logic automata: RALA. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '10* (pp. 1–6). New York: ACM. URL http://doi.acm.org/10.1145/1706299.1706301.

22. Grinstein, G., Jayaprakash, C., & He, Y. (1985). Statistical mechanics of probabilistic cellular automata. *Physical Review Letters*, *55*, 2527–2530. URL http://link.aps.org/doi/10.1103/PhysRevLett.55.2527.

23. Hutton, T., Munafo, R., Trevorrow, A., Rokicki, T., & Wills, D. (2012). Ready, a cross-platform implementation of various reaction–diffusion systems. https://code.google.com/p/reaction-diffusion. Accessed Mar 2015.

24. IEEE (1988). *1003.1-1988 INT/1992 edition, IEEE standard interpretations of IEEE standard portable operating system interface for computer environments (IEEE Std 1003.1-1988)*.

25. IEEE (Ed.) (2013). *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24–27, 2013*. IEEE.

26. Karp, R. M. (1991). An introduction to randomized algorithms. *Discrete Applied Mathematics*, *34*(13), 165–201. URL http://www.sciencedirect.com/science/article/pii/0166218X9190086C.

27. Kishinevsky, M., Shukla, S. K., & Stevens, K. S. (2007). Guest editors' introduction: GALS design and validation. *IEEE Design and Test of Computers*, *24*, 414–416.

28. Madhavan, A., Sherwood, T., & Strukov, D. (2014). Race logic: A hardware acceleration for dynamic programming algorithms. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)* (pp. 517–528). IEEE.

29. Merolla, P. A., Arthur, J. V., Alvarez-Icaza, R., Cassidy, A. S., Sawada, J., Akopyan, F., Jackson, B. L., Imam, N., Guo, C., Nakamura, Y., Brezzo, B., Vo, I., Esser, S. K., Appuswamy, R., Taba, B., Amir, A., Flickner, M. D., Risk, W. P., Manohar, R., & Modha, D. S. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, *345*(6197), 668–673. URL http://www.sciencemag.org/content/345/6197/668.abstract.

30. Misailovic, S., Kim, D., & Rinard, M. (2013). Parallelizing sequential programs with statistical accuracy tests. *ACM Transactions on Embedded Computing Systems*, *12*(2s), 88:1–88:26. URL http://doi.acm.org/10.1145/2465787.2465790.

31. Ray, T. S. (1995). *A proposal to create a network-wide biodiversity reserve for digital organisms* (Technical Report TR-H-133). Tokyo: ATR Human Information Processing Research Laboratories.

32. Renganarayana, L., Srinivasan, V., Nair, R., & Prener, D. (2012). Programming with relaxed synchronization. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability, RACES '12* (pp. 41–50). New York: ACM. URL http://doi.acm.org/10.1145/2414729.2414737.

33. Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Computer Graphics*, *21*(4), 25–34. URL http://doi.acm.org/10.1145/37402.37406.

34. Stojanovic, M. N., & Stefanovic, D. (2003). A deoxyribozyme-based molecular automaton. *Nature Biotechnology*, *21*(9), 1069–1074. URL http://dx.doi.org/10.1038/nbt862.

35. Taylor, T. J. (1999). *From artificial evolution to artificial life*. Ph.D. thesis University of Edinburgh. www.tim-taylor.com/papers/thesis/html/main.html.

36. Toffoli, T., & Margolus, N. (1987). *Cellular automata machines: A new environment for modeling (scientific computation)*. Cambridge, MA: MIT Press. URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0262200600.

37. Ulam, S. (1950). Random processes and transformations. *Proceedings of the International Congress on Mathematics*, *2*, 264–275.

38. von Neumann, J., & Burks, A. W. (Eds.) (1966). *Theory of self-reproducing automata*. Urbana, IL: University of Illinois Press.

Appendix: Source code

A.1 QLine and Line

```
/** QLine is a subatomic particle for constructing and maintaining a
    line segment, for incorporation into a complete element. It takes
    a single parameter, \c bits, which determines the atomic storage
    used by the QLine, and 2**bits is the size of the resulting line
    segment. QLine is hardcoded to grow east, but it respects and
    does not modify any symmetry settings applied to the EventWindow.
  \author Dave Ackley
  \version 2
  \license public-domain
 */
quark QLine(Unsigned bits) {
  EventWindow ew;
  typedef Unsigned(bits) Position;

  Position m_position; // Data member representing our position in the QLine
  Bool isMin() { return m_position == Position.minof; }
  Bool isMax() { return m_position == Position.maxof; }
  Void setMin() { m_position = Position.minof; }
  Void setMax() { m_position = Position.maxof; }
  /** Call to grow/maintain the line. Returns \c true if the
      overall QLine was modified in some way, else \c false */
  Bool update() {
    Bool ret = false;
    if (!isMin() && ew[1] is Empty) { // ew[1] is one site west
      Atom a = self; // This roundabout method of reproduction works around
      --m_position; // a limitation on quarks in ulam 1. We modify ourselves
      ew[1] = self; // to produce the desired offspring, saving and restoring
      self = a; // our original value before and after.
      ret = true;
    }
    if (!isMax() && ew[4] is Empty) { // ew[4] is one site east
      Atom a = self;
      ++m_position;
      ew[4] = self;
      self = a;
      ret = true;
    }
    return ret;
  }
}

/** A sixteen-site east-growing line. \symbol Ln \color #933 */
element Line { QLine(4) m_line; Void behave() { m_line.update(); } }
```

A.2 Box2

```
/** Four lines arranged in a square.
    \symbol B2
    \color #345

    \author Dave Ackley
    \license public-domain
*/
element Box2 {
  typedef Box2 Self;
  EventWindow ew;
  //typedef EventWindow.Symmetry Sym; // the 'Box' element uses this declaration of Sym
  typedef Unsigned(2) Sym;

  QLine(4) m_line; // The 2**4==16 site line that we are part of
  Sym m_sym; // The direction that line is running

  Void nextSym() { // The direction clockwise from me
    if (m_sym == ew.cSYMMETRY_270L) m_sym = ew.cSYMMETRY_000L;
    else ++m_sym;
  }
  Void prevSym() { // The direction anticlockwise from me
    if (m_sym == ew.cSYMMETRY_000L) m_sym = ew.cSYMMETRY_270L;
    else --m_sym;
  }

  Void behave() {
    ew.changeSymmetry(m_sym); // Establish our symmetry
    m_line.update(); // Build/maintain our line

    if (m_line.isMax() && ew[4] is Empty) { // Next line starts east of max
      Self b = self; // Be Like Me!
      b.m_line.setMin(); // Except you're the first of your line
      b.nextSym(); // And you're 90 degrees clockwise from me
      ew[4] = b;
    }
    if (m_line.isMin() && ew[3] is Empty) { // Previous line ends south of min
      Self b = self;
      b.m_line.setMax();
      b.prevSym();
      ew[3] = b;
    }
  }
}
```