

A common language for “programmable matter” (cellular automata and all that)

Tommaso Toffoli (tt@bu.edu) and Ted Bach (tbach@bu.edu)
ECE Department, Boston University, 8 Saint Mary’s St., Boston, MA 02215

15 March 2001

ABSTRACT

Based on extensive experience with concepts, theory, applications, and hardware and software implementations, we propose a canonical way to represent “programmable matter” structures such as cellular automata, lattice gases, and related fine-grained, uniform, massively-parallel computational substrates. Our proposal is accompanied by an “exemplar” software engine which embodies our recommendations and makes the programmable matter paradigm effectively accessible to anyone provided with an ordinary computer.

This initiative is intended to facilitate the design, as well as stimulate the exchange and dissemination, of programmable matter algorithms; among other things, it effects “technology transfer” for extensive research done in previous years by the MIT Information Mechanics group. In particular, the present approach owes much to Norman Margolus and the CAM8 project.

*Let him who hath not reinvented cellular automata
cast the first stone!*

1 Introduction

Every few months we come across an amateur who, having discovered John Conway’s wonderful game of LIFE, feels obliged to present the world with one more computer program to run LIFE simulations or with one more field report on LIFE’s “bestiary” and “ethology”.

All the while, journals dealing with condensed matter, artificial life, or complexity (to mention just a few areas) come up with papers describing sophisticated experiments in which “cellular automata”, “lattice gases”, “swarm systems”, and similar fine-grained, uniform, massively-parallel computational substrates play an essential role. Unfortunately, unlike microwave ovens or electron microscopes, these computational substrates do not come off-the-shelf: most research groups still “roll their own”. In fact, since conventional computers are not particularly efficient at fine-grained processing, to attain a tolerable level of performance in this kind of

computations there was little choice, until recently, but to run the software on a supercomputer or employ a dedicated hardware engine. In any case, one needed ample hardware resources and a substantial software effort. But this picture is changing fast.

One of us has spent a good part of his life (in collaboration for many years with Edward Fredkin and Norman Margolus) working at turning cellular automata from a parlor curiosity to a staple research tool (cf. [3, 33, 19, 20, 35, 37, 36, 17, 25]). Our point with the present paper is that this computing environment—“Programmable Matter” [35], as we shall call it henceforth—should no longer be limited only to researchers having overwhelming computing resources and the stamina to design and maintain a complex hardware/software environment. As a computational substrate, programmable matter can be made available today for evaluation and experimentation “to the masses,” in the form of software engines capable of yielding usable performance on an ordinary PC. In addition to that, in order to become a practical development tool, programmable matter must also present itself as a viable *conceptual* substrate. To this purpose, one requires depth and breadth of applications, a common notational language, a well-furnished software environment, and a wealth of working examples. As we shall see, this kind of support is also forthcoming.

Partial differential equations, though born of physics, soon escaped the physics shop floor. Since they generically deal with local interaction in continuous distributed systems, no matter where such systems occur, they literally “belong to everyone” and are suitable for all sorts of mundane applications that have little to do with physics, such as economics or epidemiology. In a similar way, programmable matter deals with local interaction in *discrete* distributed systems. No wonder it has outgrown its original “mathematical biology” cocoon and is well on its way to become one more generic “electronic data processing” tool, suitable for mundane applications such as image rendering or urban planning [40, 35, 2, 1, 22].

One might have expected that, for programmable matter, such a transfer from conceptual play to down-to-earth business would come naturally as, merely out of

ordinary technological progress, the required hardware became cheap and ubiquitous. This, however, has not happened. The last twenty years have seen the meteoric rise of the microprocessor[38], to satisfy on one hand the need for cheap embedded computing power (such as in toasters and phone cards), and, on the other, the insatiable demand of corporate enterprise for standardized desktop computing cycles (the PC, Windows, and all that). As long as microprocessors continue growing ever so more powerful and cheap, alternative architectures are bound to remain noncompetitive except in very specialized niches.

Today, paradoxically, it is dollar-for-dollar more convenient to run programmable matter on a PC hardware platform—though the latter is far from being optimal for fine-grained computational tasks—than on a special hardware engine optimized for those tasks. The greater efficiency a dedicated architecture would offer is offset by the irresistible “subsidy” that present market circumstances bestow on the microprocessor. Let us then cash in on this subsidy, then, and use PCs as a platform on which to develop awareness and appetite for programmable matter. Besides exploring applications and algorithms, this is also the right moment to build better high-level development tools, libraries, reusable “objects”, randomness generation facilities, rendering techniques, etc., as well as uniform standards, protocol, and documentation. Should the programmable-matter style of computation eventually become a veritable mass market, then the attendant economies of scale will restore the competitiveness of programmable-matter styles of hardware architecture vis-à-vis microprocessor based ones.

If scientists today can avail themselves of PCs that are powerful and dirt-cheap, it is because the office market made them so. We are looking forward to a time when programmable-matter platforms, ideal for research in material science, statistical mechanics, quantum physics, etc., will be powerful and dirt-cheap if only because they have a much wider market in mundane “killer” applications such as computer graphics and simulation.

In this paper we briefly present a project of a methodological nature aimed at providing the programmable-matter field with a “common language”, that is, common conceptual primitives, an applications programming interface (API), and a user interface. In this context, we have developed an exemplar “software engine” that responds to that API and runs efficiently on an ordinary PC, and a user-level environment built on the Python language.¹

For a review of available cellular automata implementations, programming environments, and applications we refer the reader to Thomas Worsch’s and Domenico Talia’s ample and up-to-date surveys[47, 48, 28].

¹This is an all-around interpreter and scripting/programming language[43].

2 Some preliminaries

2.1 Historical notes

Cellular automata are a discrete counterpart to partial differential equations. It is not surprising that they have been reinvented innumerable times under different names and within different disciplines. The canonical attribution is to Ulam and von Neumann[39], circa 1950. Two decades of smoldering interest were captured by Burks’ *Essays*[5]. In 1970, cellular automata achieved a surge of popularity when Martin Gardner presented Conway’s game of LIFE in *Scientific American*[11]. Connections with physics (that had already been probed by Zuse[49] and Toffoli[30]) flared up in the 80s with another impersonation of cellular automata, namely **lattice gases**[9]. Actually the lattice-gas scheme was arrived at independently, but in response to similar physical motivations, by many investigators (cf. [12, 29, 31, 7, 8, 18, 4, 34, 14, 15, 16]); its usefulness was underlined by Fredkin’s insights into reversible computation[8, 7] and a number of original applications by Margolus[18, 34], who introduced the “Margolus neighborhood”[33] and coined the term “partitioning cellular automata”. Meanwhile, Wolfram had been investigating mainly one-dimensional cellular automata in connection with statistical mechanics[45] and computational linguistics[44]. The Information Mechanics group at MIT (chiefly Edward Fredkin, Tommaso Toffoli, Norman Margolus, and, for a while, Gérard Vichniac and Charles Bennett) developed both high-performance cellular automata machines (CAM6[32] and CAM8[20]) and a vast pool of expertise in “programming” all sorts of aspects of physics in the language of programmable matter (see, for example, [41] and Mark Smith’s doctoral thesis[27]). The present initiative is a natural continuation of that program.

2.2 Legacies

It will be useful to acknowledge—and hopefully transcend—a number of minor annoyances that cellular automata have had to live with as a matter of historical legacy.

To begin with, there is the awkward term “Cellular Automata” (I remember Richard Feynman mutter “Who’d want to pronounce *that*?”²) that we owe to von Neumann himself.

To add insult to injury, cellular automata did not learn to speak physics on “their father’s lap.” Von Neumann was a most outstanding mathematical physicist with special interest in the philosophy of quantum mechanics. Thus, when he thought physics, he naturally viewed the world in terms of unitary transformations—a microscopically reversible continuum dynamics. But cellular automata were for him only a toy model for reductionistic arguments in *biology*, where he needed

²And, like with “media”, how many see “cellular automata” as a plural and have a use for its singular?

a dissipative, phenomenologically irreversible dynamics. The shortest way to achieve that was to start directly with a microscopically irreversible substrate quite unlike the physical substrate of classical or quantum mechanics. Regrettably, it took cellular automata another three decades to become expressive in the language of physics[30, 12, 33, 46].

A related issue is that, in the traditional cellular automata paradigm, local state information is conceived as *sitting* at the sites (or “cells”). By contrast, in the more modern lattice-gas paradigm, state information is thought of as *in-flight* between sites. In the latter case, a site is not where data *reside* but where they *change*. If the flow of information in the system is represented as a directed graph, the cellular automaton associates data with the *nodes* of this graph while the lattice gas associates data with its *arcs* (see §4). Note that the local transition function of a cellular automaton, of the form $F : Q^n \rightarrow Q$ (where Q is the state set of a cell and n is the number of neighbors), is typically a highly *irreversible* operation—from the new state of a single cell it would be hard to uniquely reconstruct the current state of its n neighbors.³ On the other hand, the local interaction of a lattice gas is of the form $f : Q^n \rightarrow Q^n$; if this is invertible, so automatically is the system’s global evolution. As a consequence, whereas conservation laws are nightmare to enforce in cellular automata, they become a cinch in lattice gases.

For better or worse, the distinct terms “cellular automaton” and “lattice gas” are here to stay. Even though every cellular automaton can be rewritten as a lattice gas and vice versa—and so these two computational “formats” are conceptually equivalent—nevertheless they reflect different traditions, tradeoffs, and perhaps modeling strategies. We’ll be using “programmable matter” as a generic term to denote the overall discipline in which these and other related computational paradigms (such as Programmable Logic Arrays, Quantum Dot Arrays, etc.; cf. [23]) are studied and used.

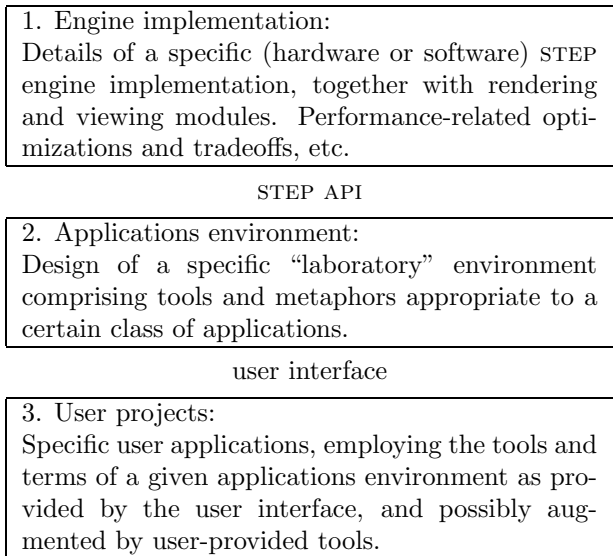
As a basis for “new design”—and by this we mean not only hardware and software but also modeling strategy, documentation, pedagogy, etc.—the lattice-gas paradigm has overwhelming advantages over the more traditional cellular automata view. In fact, we will be using this paradigm not only to describe the dynamics of particular physical or computational target systems (such as an Ising Spin ferromagnetic model or an Error Diffusion dithering algorithm), but also as the *machine language* of an ideal computational engine for programmable matter (this echoes Margolus’s view of cellular automata as the “machine language” of physics). Our architectural approach is presented in §5.

³And it is in general undecidable whether such a transition function yields an invertible mapping from global states to global states[34].

3 Objectives: design levels and interfaces

As stated in the title, our main objective is to define (and of course provide support for) a *common language* for programmable matter, so as to give this modeling discipline wider dissemination and facilitate portability of tools and applications between platforms. This common language will primarily consist of an Applications Programming Interface (API) whose data structures and procedure calls correspond to features of an abstract “programmable matter engine” called STEP (for Space-Time Event Processor). This abstract engine, which has the formal structure of a mesh multiprocessor, lends itself in a natural fashion to many different types of implementation, ranging all the way from fully custom hardware to software for a generic computer platform, from lumped (single-processor) to distributed (multi-processor, mesh, network) workload organization (see [21]).

In fact, since the STEP engine is a quite general kind of parallel processor, we anticipate its finding use in a great variety of computational contexts. Rather than interacting directly with a bare, uncommitted STEP engine (through its API), the typical user will deal with higher-level constructs appropriate to the specific applications range. We thus envisage three design levels separated by two interfaces, namely



As for level 1, we accompany the STEP API specifications with an “exemplar”—simple but reasonably efficient—software implementation of the STEP engine for the PC platform.⁴ (Note that a hardware engine that concretely embodies the STEP abstraction already exists, in the form of the CAM8 lattice-gas multiprocessor[20]; however, even though several CAM8 units exist and are in use at a number of laboratories, this machine is the outcome of an academic project and its availability and

⁴This is written in C with provisions for assembler optimization of critical loops. The initial release runs under Linux with X display; a Window release is forthcoming.

support are limited.⁵⁾

As for level 2, we offer a preliminary version of an all-around programmable-matter “laboratory”, called SIMP. This is an environment, based on the Python language, for describing, interactively running, and scripting all sorts of programmable matter applications or “experiments”. SIMP provides prefabricated *modules* containing user-level data structures and facilities for rapidly implementing programmable-matter models. Its modeling strategy reflects, on a higher abstraction level, the basic signal-and-event primitives that make up the “machine language” of STEP.

The Python language on which SIMP is based addresses in a modern way the requirements of users who do not have the patience to re-invent the wheel every day: they will gladly base their designs on components that are ready-made (and *professionally* made). On the other hand, they want a programming environment based on a flexible, general-purpose computer language, one which can serve all of their needs in different walks of life and that they can “grow” with. Incidentally, similar requirements were addressed, in a more modest context, by the Forth-language environment used to manage the CAM8 engine.

Finally, level 3 starts out with a number of d]mos and experiments, many transliterated (or at least paraphrased) from the CAM8 context to the SIMP environment, i.e., exploiting the SIMP user interface. Eventually this level will contain the users’ new creations—the bottom-line applications of programmable matter.

4 A simple lattice gas

We assume the reader to be familiar with the cellular-automaton format of computation. Briefly, with reference to Fig. 1, which depicts the geometry of a simple one-dimensional system, we have a uniform array of storage elements, or *cells*, indicated by δ (for “delay element”). Cell data are continually recirculated through each cell, passing at each iteration through a *transition function* F . Beside the data from the cell itself, this function receives, as additional inputs, copies of the data from some *neighbor* cells (in the figure, the right and left cells). For the game of LIFE we would have a similar picture, but with a two dimensional array and eight neighbors from eight compass points (N S E W NE NW SE SW).

By unfolding the iteration loop, the same cellular automaton can be represented as the loop-free combinational network of Fig. 2 (this latter representation is really the primary one from a conceptual viewpoint).

⁵Moreover, the huge performance advantage that CAM8 originally had over traditional computer hardware in programmable matter applications is being steadily eroded by the normal effect of Moore’s law. Only by becoming commercial and undergoing periodic respins will an architecture like CAM8 retain its advantage. It is hoped that the present effort will pave the way for such technology transfer.

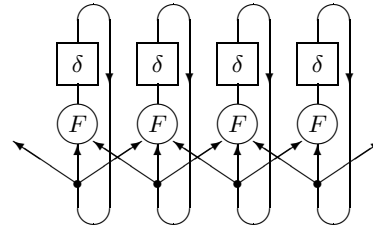


Figure 1: A cellular automaton as a sequential network. We illustrate the case of one dimension and first (left and right) neighbors.

The delay elements δ have gone, since they were merely bookkeeping devices, used to tell where to cut open the loop of Fig. 1 so as to obtain the “unit strip” out of which Fig. 2 is generated.

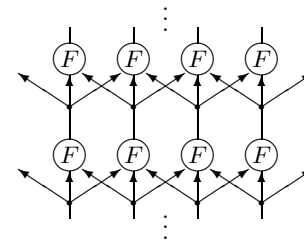


Figure 2: The cellular automaton of Fig. 1, re-expressed as a combinational network by “unrolling” the loop of Fig. 2.

In a lattice gas, on the other hand, what we find at the sites is not data storage elements but spacetime *events*, i.e., junctions where *signals* come together, interact, and then depart. Signals with definite values exist only *between* sites rather than *at* sites. In Fig. 3 we have a two-dimensional system, with four signals arriving from the four compass points and four departing at every site. Since this is actually a sequential-network representation like that of Fig. 1 rather than a combinational one, a delay element δ should be imagined as straddling each arc in the vicinity of each label.

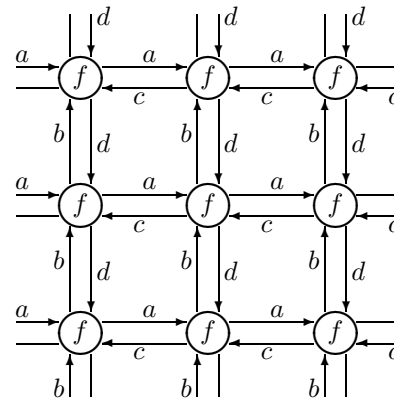


Figure 3: Geometry of a simple, two-dimensional lattice gas such as HPP.

The same system is expressed as a *spacetime diagram*—or a loop-free combinational network—in Fig. 4.

With respect to Fig. 3 we have one more dimension, representing time; in this fashion, it is easier to see that, with respect to a site, incoming arcs and outgoing arcs belong to different moments of time. Again, the δ s have gone. Signals travel in inertial motion from site to site (look at the a signal, for instance, which proceeds in the $(1, 1)$ direction both on entering and leaving a site).

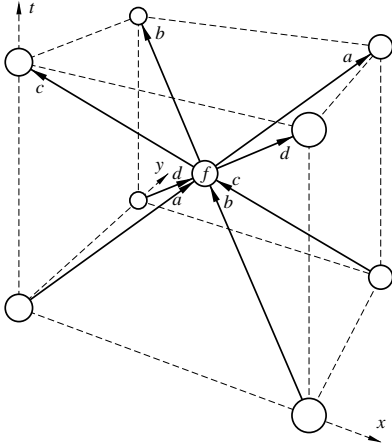


Figure 4: Spacetime layout of a simple lattice gas. This structure is iterated over spacetime.

At this point we can think of a specific interaction rule for f . In the HPP gas[12], the arcs are thought of as “tracks” for “particles”, a value of 1 representing the presence of a particle, and 0 an empty track. No more than one particle is allowed on each track. With reference to Fig. 5, particles crossing one another’s paths go through a site unaffected, while particles colliding head-on are scattered at right angles (provided that the new tracks are at the moment empty and thus can accept the scattered particles—otherwise no scattering will take place).

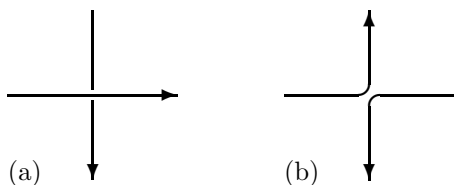


Figure 5: In the HPP lattice gas, particles crossing one another’s paths (a) go through a site unaffected, while particles colliding head-on (b) are scattered at right angles.

The collision rule f for HPP can be written as a lookup

table:

in	out	in	out
$abcd$	$abcd$	$abcd$	$abcd$
0000	0000	1000	1000
0001	0001	1001	1001
0010	0010	* 1010	0101
0011	0011	1011	1011
0100	0100	1100	1100
* 0101	1010	1101	1101
0110	0110	1110	1110
0111	0111	1111	1111

Note that in only two cases (marked with an asterisk) out of sixteen does a nontrivial interaction take place; in all other cases, each of the four signals proceeds undisturbed. Since the function f represented by this table is invertible, a spacetime history may be extended backwards as well as forwards in time.

As soon as the numbers involved become large enough for averages to be meaningful—say, averages over spacetime volume elements containing thousands of particles and involving thousands of collisions—a definite continuum dynamics emerges. And, in the present example, it is a rudimentary *fluid* dynamics, with quantities recognizably playing the roles of density, pressure, flow velocity, viscosity, speed of sound, etc. Fig. 6 illustrates sound-wave propagation in this model. Note that, even though the microscopic interactions only display a limited form of rotational symmetry (namely, invariance under quarter-turn rotations), the speed of sound in the HPP gas is fully isotropic.

Unlike sound speed, however, sound attenuation is *not* isotropic in the HPP model. It turns out that, besides conserving energy and momentum, HPP separately conserves the horizontal component of momentum on each horizontal row and the vertical component on each vertical column. These *spurious* conservations (they have no counterpart in ordinary physics) lead to significant departures from the behavior one would expect from a physical fluid. The slightly more complicated FHP lattice gas model[9]—which uses six rather than four particle directions (always in two dimensions)—gives, in an appropriate macroscopic limit, a fluid obeying the well-known *Navier-Stokes* equation, and which is thus suitable for modeling actual hydrodynamics (see [13] for a tutorial). Analogous results have been obtained for three-dimensional models[10].

5 Engine interface: the STEP API

The HPP lattice gas will supply the paradigm for the abstract STEP engine. (For simplicity, we shall often speak as if the system extended indefinitely in both the x and y directions. In practice, space may be “wrapped around” as a torus. If we call r and s the lengths of great circles of this torus, it will be sufficient to interpret all address arithmetic, on vectors of the form (x, y) , as performed modulo (r, s) . Alternatively, the array may

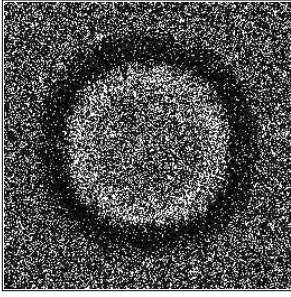


Figure 6: Wave propagation in the HPP lattice gas. Note the emergence of circular symmetry.

have actual edges, in which case explicit provisions will have to be made as to what must happen there.)

In our abstract engine, instead of storing data “by the site”—grouping close together in the storage medium all signals that are close to one another in the physical system—we shall store data “by the signal”. That is, we shall have four separate bit arrays or *layers*, one consisting of all signals of type a (cf. Fig. 3), one of all signals of type b , and so forth. Each layer will be spatially organized like the entire space (same number of dimensions, length along each dimension, etc.). However, the layers will not be permanently ganged together, but will be able to shift the origin of coordinates with respect to one another along any dimension. (In terms of implementation, they will have independently movable read/write “heads”—indexing pointers, if in software, or memory controllers, if in hardware.) Given an arbitrary relative positioning of the four layers, we may imagine stacking them on top of one another in good registration, so that the four signals impinging at a certain moment on a given site will all have the same (x, y) coordinates in the respective layers, and will make up, as it were, a four-bit *pile*.

The mechanics of the HPP gas is then simply to iterate a *step* consisting of two stages:

Interaction Apply function f , as described in (1), to each site, using as an argument the four-bit pile at that site and then replacing this pile with the four-bit result of the lookup.

Transport Independently shift each of the four layers as a whole, one along the a arc, one along b , and so forth. We still end up with a four-bit pile at each site, but with a new make-up.

Note that, at the interaction stage, piles are processed independently of one another, so that the *order* in which they are updated is irrelevant. One could have several copies of the lookup table (or whatever kind of processors are used to implement the interaction function f), do some (or all) of the processing in parallel, whether in lockstep or asynchronously—no coordination is needed between these processors. Moreover, though the identity of the four data components (which bit is an a , a b , etc.) is important for the interaction, the direction and

the velocity of the corresponding signals is irrelevant. The interaction processor does not have to know from what direction and from how far the a bit has come. In this sense, the interaction stage performs a pointlike, *geometryless* operation. Any references to the geometry of the array and of the signals are relegated to the transport stage.

At the transport stage, note that the shift performed on each layer is a *uniform* and *data-blind* operation (all bits are translated by the same spatial offset, quite independently of their positions and values); thus, it becomes possible, in a suitable implementation, to replace this operation by one that just shifts the frame of reference by adjusting a pointer rather than bodily moving the data themselves. At the transport stage, it is the layers (rather than the piles) that are processed independently of one another; no coordination between the different layers is required. *No pieces of data are actually modified or even looked at!* The whole transport choreography is just a collection of separate and quite trivial address-arithmetic operations. Any references to the values of the data themselves is relegated to the interaction stage.

Note that, even though we used a specific example, we have been describing a strategy for implementing a *generic* lattice-gas dynamics. In fact, the nature of the above prescriptions is quite independent of the number of dimensions, the number of signals interacting at a site, the state set of each type of signal, the “crystallography” of the event/signal mesh, and the nature of the interaction rule. Thus, the interaction and transport stages detailed above for the HPP gas can be immediately generalized to an arbitrary lattice gas.

To arrive at the basic specifications of the STEP engine, we start from the lattice-gas paradigm and introduce one restriction and one generalization:

Restriction All elementary signals in STEP are *binary* signals. Signals with more than two states are synthesized by ganging together binary signals. Thus, if our target system has a three-state signal (with values, e.g., $-1, 0, +1$), we will represent it as a pair of binary signals; this gives us four states, of which three will be used for the signal and the fourth will remain unused.

This restriction, which simplifies the STEP interface, does not entail any loss of generality.

Generalization The number of spatial dimensions and the number of elementary signals of a STEP engine are established at the moment the engine is created. However, the dynamics of these signals, as specified by the interaction and transport stages, may freely be changed from step to step, possibly as a consequence of external, just-in-time decisions. Specifically, at every step we are allowed to use a *different* interaction function f . Note that the *format* of the interaction function f —that is, the “length”

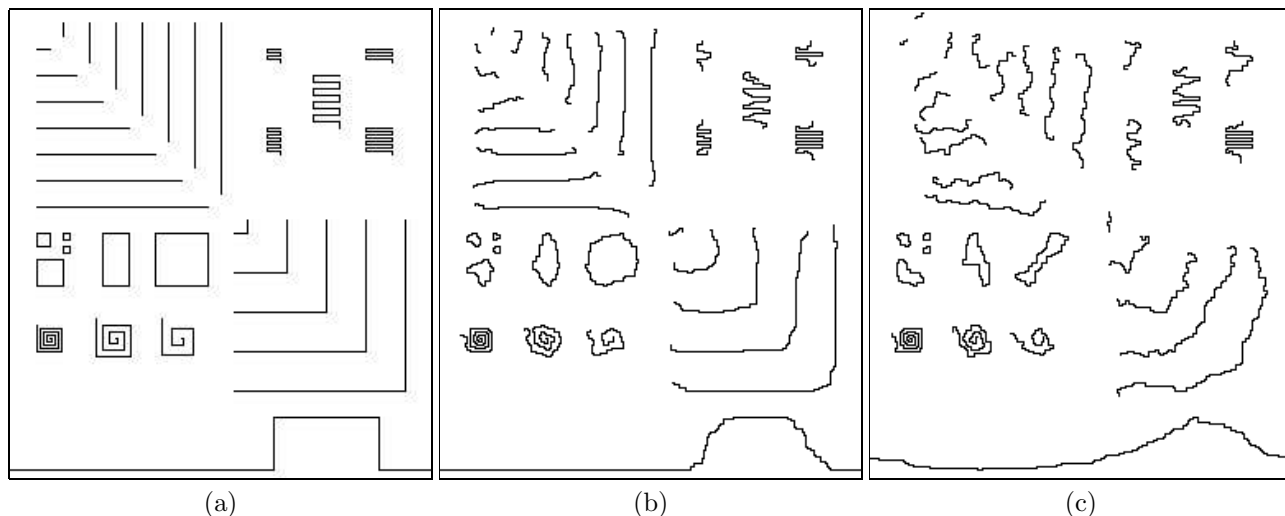


Figure 7: (a) A collection of polymer strings in quite unlikely (much too regular) configurations. (b) Some time later, when thermal agitation has had a chance to introduce some disorder. (c) Much later, when at least the shortest chains have reached typical (equilibrium) configurations.

and the “width” of the interaction table—will remain the same,⁶ but the mapping specified by the function—that is, the *contents* of the table—may be assigned afresh at every step (during the entire step, of course, it will remain the same for all sites of the array). Likewise, at every step the collection of vectors that specify the shifts for the different layers, i.e., the signals’ directions and velocities (and thus ultimately the mesh’s spacetime interconnectivity), may be specified anew.

This generalization is what makes programmable matter truly programmable. In fact, by stringing together steps with different interaction and transport characteristics it is possible to synthesize arbitrarily complex “macro-steps”.

6 An example: polymer diffusion

We’ve claimed that the STEP “machine architecture”, though exclusively based on lattice-gas primitives for data transport and interaction, yet provides very general programming resources. To illustrate this, we will show how to go about programming in STEP a dynamics, namely, *polymer diffusion*, that does not obviously call for a lattice-gas implementation. First we shall describe a generic modeling approach to polymer diffusion (for simplicity, in two dimensions); we will then implement this approach by means of STEP machinery. (The approach presented here is paedagogically more transparent, as well as more general and flexible, than the related one described in [27].)

By *polymer* one usually means an assembly of similar objects held together by forces that strongly encourage the formation of linear chains. In addition, weaker at-

tractive or repulsive forces are present between nonadjacent chain units, making certain chain configurations more likely than others. The equilibrium configuration of a polymer melt (a collection of chains) is a compromise between the ordering tendencies of those forces and the disruptive action of thermal agitation; thus, a stretched chain will tend to contract—and a neatly packed configuration to expand—into a loose clump of intermediate “kinkiness”, as shown in Fig. 7. We’d like to set up a dynamics exhibiting this behavior in a qualitative way and possibly capable of providing quantitative information. Though the chain links are in first approximation unbreakable, we’d also like to be able to play with dynamics where, with certain probabilities, links may break or be formed.

Let us consider an assembly of identical particles occupying some of the nodes of a square grid, as in Fig. 8a. In order to interpret this assembly as a polymer we’ll treat two particles as **linked** if they are adjacent on the grid, i.e., if they are directly joined by a grid arc, as indicated in Fig. 8b.⁷ The arc joining the two particles will be called a **link**.

Note that in our system links are not assigned as *separate data objects* but are implicitly given by the relative position of the particles. According to the number of links originating from a particle, one may characterize the latter as

- No links: An **isolated** particle.
- One link: An **end-chain** particle.
- Two links: A **mid-chain** particle.
- Three or four links: A **branch-point** particle.

⁶Since, in a lattice gas, as many bits come out of an event as go in, the number of binary signal per site is a conserved quantity.

⁷If the grid nodes are identified with the elements of \mathbb{Z}^2 , a link is a pair of nodes which differ by ± 1 in one coordinate and by 0 in the other.

Fig. 8 provides examples of all four kinds of particles.

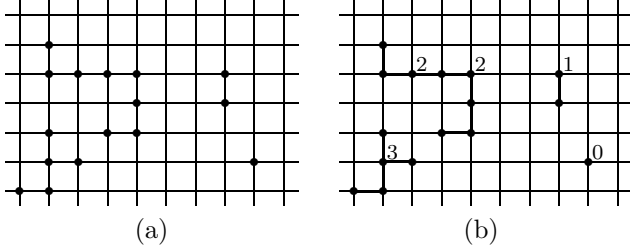


Figure 8: (a) A number of particles scattered on the nodes of a square grid. (b) Adjacent particles are implicitly connected by links; note the presence of isolated (0), end-chain (1), mid-chain (2), and branch-point (3) particles.

A sequence of particles in which all elements except the first and the last are mid-chain particles will be called a **chain** (at the two ends of a chain one will find either an end-chain particle or a branch-point particle). We are going to define a dynamics, suitable for a polymer, where particles have a certain degree of mobility but nonetheless chains retain their identity. Note that, in our elementary model, the angle between two consecutive links can only have the discrete values of -90° , 0° , and $+90^\circ$ (“turn right”, “go straight”, “turn left”).

In our dynamics, individual particles will be allowed to move to a nearby grid position only if they can do so without breaking a link or creating a new one. To this purpose, the recipe for a move must be allowed to examine, depending on the proposed move, some of the first neighbors of the particle. Fig. 9 shows some proposed moves and which of those can be accepted. Particle 2 can move as proposed; particle 3 can’t, as that would create a link with 1. Particle 8 can “flip” about the 7-9 diagonal (this will be the typical move for polymer diffusion); particle 5 cannot move to any of the proposed sites as that would break one or both of its links with 4 and 6.

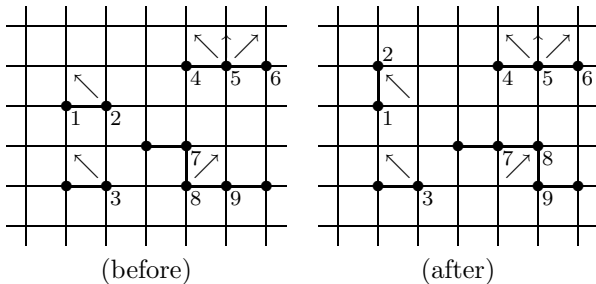


Figure 9: (a) Proposed moves. (b) Particle 2 can move as proposed; particle 3 can’t, as that would create a link with 1. Particle 8 can “flip” about the 7-9 diagonal; particle 5 cannot move to any of the proposed sites as that would break one or both of its links with 4 and 6.

Given any square of the grid, a particle may be located in any of its four corners. In general, the only kind of move that the particle may be proposed will be to traverse the square, as indicated in Fig. 10 and Fig. 11.

The proposed move will be carried out if the destination position is unoccupied and if no links would be created or destroyed. It is easy to verify that this happens if and only if the sites marked with a hollow circle are empty. Note that the occupation state of the sites marked with a cross is irrelevant, since any links between the particle and these sites will be preserved by the move.

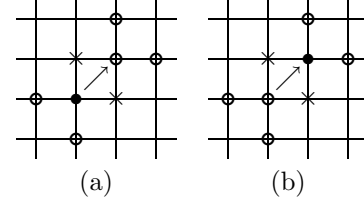


Figure 10: In order to preserve links, a proposed move across a square, from (a) to (b), can be honored only if the sites marked with a hollow circle are vacant. The occupation of the sites marked with a cross is irrelevant, since any links between the particle and these sites will be preserved by the move.

Since the occupation of the sites marked with a cross is irrelevant, the above rule (Fig. 10) can be applied not only to particles in mid-chain position, but also to end-chain or isolated particles. It also applies in a vacuous way to branch-point particles, since the latter do not satisfy the prerequisites of Fig. 10 and therefore are not allowed to move at all.

We can now make a remark that *drastically simplifies the characterization of what particles can move*. Let us think of a move as a *swap* of contents between two sites that face one another across a diagonal (in Fig. 10, these sites are the beginning and end positions of the particle). If the sites across this diagonal are *both empty* or *both occupied*, then swapping them is effectively a no-op, and thus is always allowed. If only one is occupied by a particle, then both the move indicated in Fig. 10 and the opposite move, obtained by running it backwards, are permitted. Therefore, our rule can be more simply stated as follows:

POLYMER DIFFUSION LOCAL UPDATING RULE. *In configurations like those of Fig. 11, where the four circled sites lying on the two sides of a diagonal are empty, it is permitted to swap the contents of the two sites across the diagonal. For the sake of this permission, the contents of all other array sites is irrelevant.*

Either of the two diagonals of a grid square may thus act as the pivot for a swap operation, and in this role we shall call it a **locus** of activity. In what follows, we shall use the “dogbone” stencils of Fig. 12 to highlight specific loci where we intend to apply the above rule.

Now that we are in possession of a local “swap” primitive consistent with the polymer interpretation of the array’s contents, it is our responsibility to appropriately iterate this primitive in space and time so as to synthesize a plausible polymer dynamics.

Note that moving a particle at one locus will change the site occupation pattern in which move permissions for

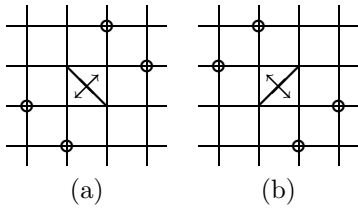


Figure 11: The basic rule for polymer diffusion is simply, *Feel free to swap sites across a diagonal if the circled sites in the indicated positions are empty*. In (a) and (b) we explicitly show the two possible diagonal orientations; the sites to be swapped are indicated by the double arrows.

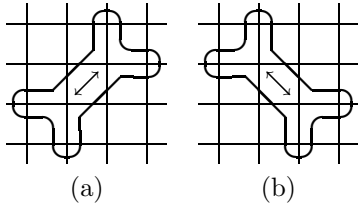


Figure 12: The “dogbone” stencils will be used to highlight specific places where we intend to apply the polymer swap rule of Fig. 11.

particles at neighboring loci are evaluated. For example, if we look at the partially overlapping dogbone stencils in Fig. 13a, either stencil highlights a legitimate move; however, once particle 1 is moved, particle 2, who used to sit on a chain elbow, will now sit on a straight segment and in this position (like particle 5 in Fig. 9) will not be able to move. Applying both candidate moves at once would break the chain (Fig. 13b). On the other hand, the third dogbone can be applied independently of the first two since it does not encompass, in its four sensitive “horns”, any sites that might have been affected by moves in the other two dogbones.

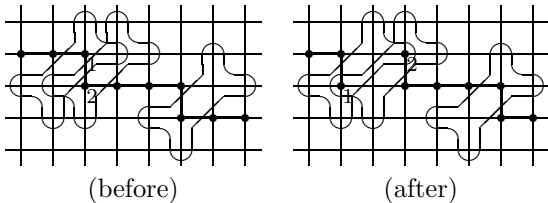


Figure 13: As explained in the text, swap operations that are individually permitted by the “dogbone” stencils can be carried out jointly only if the outcome of one does not modify the “permission context” of the other. Here, the two moves highlighted by the overlapping dogbones are not independent: if both are carried out at once a spurious chain break between 1 and 2 occurs as shown.

To avoid interferences between moves like the above, of course the most conservative strategy would be to swap at only *one locus at a time* and re-evaluate all permissions after each move. A substantially equivalent strategy would be for the updating of each potential swap locus to be *enabled* for an instant at random times, with an independent Poisson distribution at each lo-

cus. Strictly speaking, with this “Poisson updating” [33] there is only a vanishing probability that two interfering swap operations (cf. Fig. 13) will be attempted at *exactly* the same time. But, of course, in any physical implementation of the updating algorithm, a move will take some finite time to complete, and then the chances of interference between moves are no longer vanishing.

A strategy for massively parallel updating, having essentially the same effect as one-locus-at-a-time, and thus avoiding interference between moves, would be to position a number of **event processors** over the array (a) *randomly* but (b) *avoiding stencil overlap*.⁸ Each processor would be responsible for applying the updating rule to one dogbone stencil, and all processors would work concurrently. To keep the dynamics running, one would make successive passes over the array using a new random placement of processors at each pass.

7 STEP implementation of the model

From a practical viewpoint, randomly positioned event processors—whether hardware or software—are bound to be an implementation nightmare. Here we shall show how to approximate this strategy as closely as desired by using a collection of *uniformly arrayed* event processors together with a uniform signal-transport scheme, that is, by means of a STEP engine like that sketched in §5. To make the following discussion simpler, it will be convenient to use, rather than a dogbone, stencils encompassing *both* of the dogbones centered on a particular grid square, as in Fig. 14a, yielding the “swiss cross” tile of Fig. 14b. It will be up to a particular implementation of the rule to decide whether one, the other, or both swaps within a tile are attempted. Note that the swaps associated with the two diagonals are independent of one another and can be carried out concurrently.

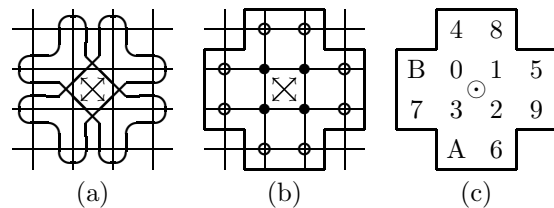


Figure 14: By merging the two dogbone stencils (a) centered on a square we obtain the “swiss cross” tile (b). Here the candidates for swaps are the four sites in the middle (dotted), while the remaining eight sites (circled) provide the permission context for those swaps. In (c) we number the sites within a tile according to the STEP layers they will be stored in; we also place a reference mark in the center of the tile.

Our intention is to have the tiles play a *dual function*,

⁸Requirement (b) makes it impossible for the random placement (a) to be truly independent between sites. However, independence may be approximated by making the placement so sparse that the need to override (a) in favor of (b) would only occur rarely.

that is, to be involved both in data interaction and in storage/transport. To this purpose, we shall uniformly cover the whole grid with tiles, as indicated in Fig. 15. Because of the way grid sites are embedded in the tiles, the tile array will use different coordinate axes than the grid and coarser units along these axes. (We use the symbols X and Y for the tiling axes and x and y for the grid axes; in the CAM8 tradition, we have y point downwards.)

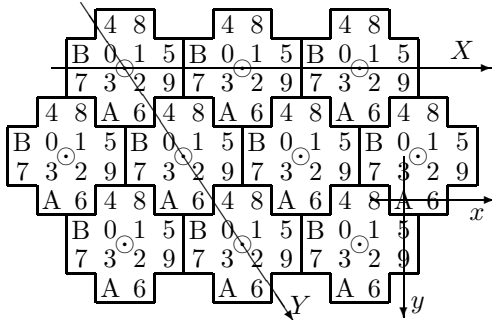


Figure 15: .

Now, recall from §5 that in HPP we had viewed the four arcs that impinged on a node—one from each layer—as a *pile* of bits, to be operated upon as a unit and turned into a new pile by the interaction function. The twelve polymer sites within a tile interact in a similar way. Each tile is served by an *event processor*, all processors acting concurrently. At each updating step the twelve site bits under purview of this processor are picked up from the layers and fed as inputs to the processor; this returns as outputs twelve new bits which are dropped back, at the original positions, into the layers. In our case, the interaction will be a permutation of a tile’s four center sites⁹ made conditional on the contents of the eight peripheral sites. This is schematically indicated in Fig. 16; note that the peripheral sites, though fetched for inspection, are returned unchanged.

In the HPP gas we had viewed the arcs going North, South, East, and West as four different kinds of “materials” to be kept in four distinct *layers* for storage and transport purposes. In a similar way, here we shall treat the twelve sites in a tile as twelve different kinds of data, to be stored in different layers numbered 0 1 2 3 4 5 6 7 8 9 A B, as indicated in Fig. 15.

When the event processor array—which henceforth we’ll call the *STEP mesh*—is positioned over the grid as shown in Fig. 15 and performs an updating pass, it actually processes only a fraction of the potential swapping loci, i.e., only the grid squares currently at the center of a tile, as in Fig. 17. The event processors make up, as it were, the “teeth of a rake” (a two-dimensional rake, in our case) that “stir up” the grid’s contents only at certain places, in a regular spatial pattern.

⁹Specifically, we are going to swap particles. However, with this tile, also more complex permutations of the particles at the four central sites become possible.

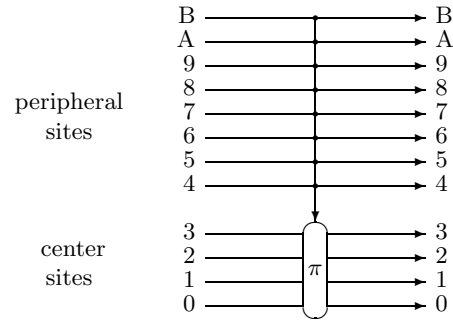


Figure 16: Schematic functional behavior of the action of an event processor on a pile. The eight peripheral sites 4 5 6 7 8 9 A B are inspected to determine how particles in the four central sites 0 1 2 3 should be rearranged. Each different combinations of values on the peripheral sites (hinted at by the “wired AND” line that joins signals 4 through B) will select a different permutation π to act on the central sites 0 through 3.

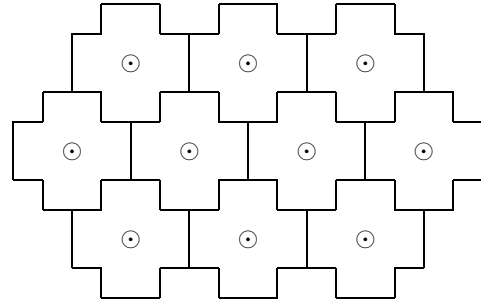


Figure 17: Currently active loci (marked by circles) for one position of the event processor array on the grid.

In order for all of the grid loci to have a chance to be activated at some time or other, we’ll have to “drag” the rake across the grid, or, equivalently, move the grid under the rake. No matter in what direction and how far the rake is translated to move to a new position, its teeth will retain the same relative spacing, each time covering the grid with a similar, but differently positioned, “interaction activation” pattern (Fig. 18).

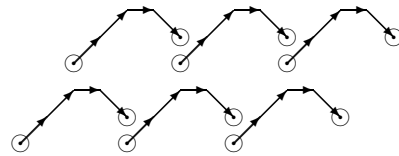


Figure 18: Successive positions on the grid of the event processor “rake”. The first and last activation patterns are indicated by circles.

Since a tile encompasses *twelve* grid nodes but only activates interaction of particles across the square at its center, there are twelve distinct positions in which the STEP mesh can be laid over the grid.¹⁰ An appropriate

¹⁰Note that, in a square grid, there is a one-to-one ratio of nodes to squares. The ratio between nodes and “plaquettes” may be different with other types of grid.

activation schedule will decide in what sequence (regular or irregular, deterministic or random) one shall step through these twelve positions in order to repeatedly apply to the entire grid the polymer interaction rule (cf. schedule). What schedule is most “appropriate” depends, of course, on the specific model one is running and on the objectives one has in mind for it (e.g., whether we are more interested in faithfulness of the microscopic dynamics or in long-run statistical accuracy). Let’s us consider for the moment one of the simplest activation schedules. In Fig. 19, the array of event processors is repeatedly moved one grid position down and to the right between applications of the interaction rule. For the first eleven steps, the processors encounter “virgin ground”; on the twelfth, each processor encounters a site that had already been touched by some other processor twelve steps before. In other words, every twelve steps the “rake” will have stirred once the entire grid, giving each locus a chance to rearrange the particles surrounding it.

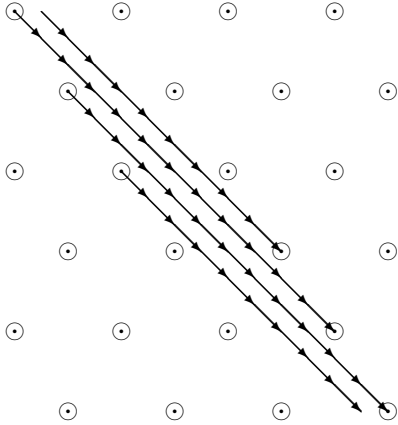


Figure 19: Starting from a given position, twelve successive translations of the processor array return this array to a position indistinguishable from the original, having touched all grid sites once in the mean time. The arrowheads indicate the successive positions of two processors, and coincide with grid sites.

In conclusion, we have outlined a (somewhat naive) twelve-step lattice-gas “macro” that will sweep over the grid, activating the prescribed interaction rule once at each locus and avoiding updating interference between adjacent loci. By iterating this macro we’ll be subjecting the system to a stylized polymer dynamics.

In the rest of this section we’ll explain in more detail the transport phase (§7.1) and then examine in a more critical way the activation schedule presented here and suggest more effective alternatives for it. Finally we shall make a few remarks about the interplay of global topology, local topology, and group theory in determining appropriate geometries for grid and mesh.

7.1 Transport details

Recall from §5 that, in a STEP model, the storage location of a state bit is identified by two kinds of infor-

mation, namely which *layer* (in our case 0 1 . . . B) and which *pile*. In turn, the location of a pile will be a set of spatial coordinates (in our case, a pair (X, Y)) in the event processor mesh, as in Fig. 15.

To make the event processors mesh move one position down-and-right on the grid as in Fig. 19 is tantamount to moving the grid one position up-and-left, as in Fig. 20.

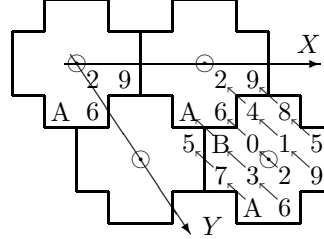


Figure 20: To make the event processors mesh move one position down-and-right on the grid as in Fig. 19 is equivalent to moving the grid one position up-and-left as indicated by the small arrows.

We shall now resolve such a motion, for each grid bit, into motion between layers and motion between piles. With reference to Fig. 20, we see that a bit on layer 9 will move to layer 1 of the same pile, while a bit on layer 4 will move to layer 2 of a different pile, namely, one that has the same value for X but one unit less for Y . The complete transport table will consist of two columns, one indicating to what layer a certain bit will move, and one indicating whether it should remain in the same tile or end up in an adjacent one:

from layer	to layer	mesh offset $(\Delta X, \Delta Y)$
0	6	(0, -1)
1	4	(0, 0)
2	0	(0, 0)
3	B	(0, 0)
4	2	(0, -1)
5	8	(0, 0)
6	3	(0, 0)
7	5	(-1, 0)
8	9	(0, -1)
9	1	(0, 0)
A	7	(0, 0)
B	A	(0, -1)

Note that the two operations, shuffling layer addresses within a pile and shifting pile addresses within a layer, *commute*. Therefore, the transport operation (change layer and mesh coordinates of a bit) indicated by the above table can be split into two separate operations, one performed within piles and one within layers.

Even though in our model this shuffling of bits within a pile is conceptually part of the transport stage, in the STEP architecture it belongs to the interaction stage, and indeed can be merged with the interaction function proper as shown in Fig. 21. During the interaction stage one might do two separate passes over the grid, one to carry out the interaction proper and one to carry out the shuffle. In practice, however, it may be more efficient to

run the two tasks in the same pass; for example, if the processor realizes the interaction function by means of a lookup table, the shuffle can be compiled directly into this table so that a single lookup will be sufficient.

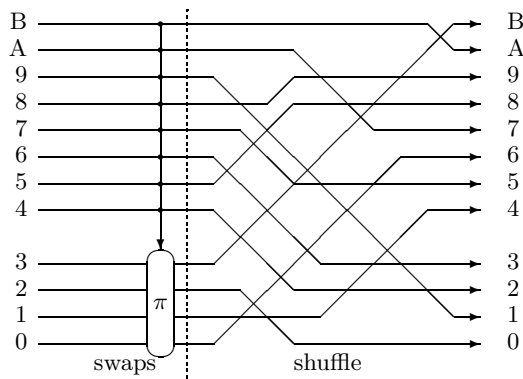


Figure 21: In STEP, the shuffling of bits between layers within a pile, preliminary to bodily shifting the layer themselves, is realized as an “adjunct” of the interaction stage (the permutation network on the right) rather than part of the transport stage.

7.2 Activation schedule: coherence and randomness

In the foregoing discussion, the discreteness and uniformity of the underlying grid came with the decision to go for a programmable matter model. The fact that the interaction rule had a scope of six or twelve neighboring grid sites (respectively for the dogbone stencil and the swiss-cross stencil) reflected the nature of the polymer dynamics. On the other hand, it was an implementation artifact that the event processors were *coherently* arrayed over the grid and that the resulting processor mesh was made to advance over the grid with rectilinear and uniform motion—yielding a *regular* twelve-step activation schedule for the grid sites.

If our intent was to approximate the independence between sites characteristic of Poisson updating (§6), with this schedule we are probably falling short of the goal since we introduce patently spurious correlations between spacetime events. For example, loci are activated in a very short and regular cycle, and so there is the risk that swaps performed during one part of the cycle will coherently be undone—Penelope-like—during another part, yielding no net evolution.¹¹

There are two fairly obvious (and fairly simple) ways to attempt to destroy the coherency of the processor mesh as seen by the grid.

- In Fig. 19, the mesh repeatedly traverses in an identical order the twelve relative positions it can take

¹¹For a similar situation in a simpler setting, suppose that three moves—“hop right”, “hop left”, and “rest”—are available in order to make a particle diffuse along a one-dimensional array. The regular sequence “...right-left-right-left-...”, though keeping the particle in motion all the time, effectively holds it pinned in place.

with respect to the grid. But one can have a random number generator decide what position to visit next (this is a fairly inexpensive device, as only one piece of randomness is used up at each macro-step). Now, neighboring grid sites will be hit at random, and thus to a very short stretch of polymer the updating will “feel” Poisson-like.

To see a potential shortcoming of this approach, assume that at one moment we have two short, isolated, *identical* polymer chains separated by an *integral number of mesh units*. They will be hit “at random” and thus will individually display a believable dynamics, but they will be both hit by exactly the *same* sequence of stencils, and thus will evolve in an *identical* way; in most realistic situations, this would be viewed as an artifact.

- At every step, for each of the event processors—and thus for each of the loci indicated in Fig. 17—we can use a random bit to decide whether the proposed interaction will be carried out or instead inhibited. This will “knock off” tiles at random; if the damage to the mesh is extensive, what will be left will appear a quite random fabric.¹² At any rate, now identical chains will “misfire” at different moments at any given place, and thus their histories will diverge.

The present expedient, whose action is fairly complementary to the first, is much more expensive in that randomness is consumed by the event rather than just by the step. However, one can rely on the fact that in most modeling situations large parts of a large system can be treated as random number generators for one another. Thus, suppose that we add to the mesh an extra bit layer containing a random pattern generated *once and for all*, and we use this pattern as an activation mask. On the first step on which we use it, this pattern is of course truly novel. In preparation for each successive step, we translate the whole random pattern by a large, random number of mesh positions. Thus, we reuse the same pattern, but any given portion of the pattern will be reused by a *different* portion of the grid, so that grid pattern and random activation pattern will be “new” to one another. Indeed, this is method used (together with the above random positioning of the mesh over the grid), to obtain the polymer dynamics of Fig. 7. This approach has been used extensively in CAM8 experiments.

A further refinement would be to have the random pattern be modified locally by the dynamics on the basis of the current contents of the grid just as the latter is modified by the former. In this way, the original randomness would be continually “stirred”

¹²Though, of course, the loci that remain will be spaced by multiples of mesh units rather than of the finer grid units.

by each use and remain in effect incoherent or “im-predictable” to the polymer.

Innumerable other approaches are possible offering complements and alternatives to the above. Transition probabilities related to the local energy contents via a Boltzmann factor require a source of randomness that is tunable on a moment-by-moment and locus-by-locus basis; Smith[26] describe a canonical way to synthesize these just-in-time probabilities from a small set of spatially uniform distributions. The real challenge is not technological but conceptual: What is it that we want and why? How can costs and benefits be compared? What can we afford to get away with? In short, What difference will it make? These are questions that involve mathematical, physical, and computational-science maturity more than technological know-how.

8 Programmable matter lab: the SIMP user environment

As mentioned in §3, the typical programmable-matter user will not directly employ the STEP API but will work in a higher-level production environment—a “shop floor” with machine tools appropriate to a certain class of applications and a certain category of users.

For our own convenience, we have been developing—and will be happy to share—an environment suitable for experimentation and prototyping. This applied-research “laboratory”, called SIMP, consists of a number of essential engine “cores” and other facilities for running, rendering, and displaying programmable matter—the whole embedded in the general-purpose scripting/programming language Python. In this way the user has a smooth transition path between what is provided and what must be developed, between tools that are specific to programmable matter and others that are generic trade tools in computer-based experimentation. An illustration of the basic capabilities of SIMP is provided by the program of Fig. 22, which encodes a STEP implementation of the polymer model discussed above. The program is divided into a number of small sections; we’ll provide a section-by-section commentary with occasional digressions.

In the Geometry section we assign the basic topology/geometry of the STEP mesh. This a torus, for which we specify the length of each dimension (and implicitly the number of dimensions) by means of a *dimension vector*, in this case `[X,Y]`. As we shall see in §9, it is possible to specify a torus described by a more general dimension *tensor*, but SIMP will give the correct default interpretation if only a vector is specified as in this case. Even though in this example we use only binary signals, SIMP supports signals with more than two states. In the Signals section we use a *signal allocator* that allocates binary layers to signals as appropriate, matching the layers’ numeric addresses within the pile structure with the mnemonic names given to signals by the user. Thus,

```
#----- Geometry
X = 50; Y = 100;
dimen = [X,Y]      # Number and size of dimensions

#----- Signals
StartSigAllocation()
p0 = NextSig(1); p1 = NextSig(1)
p2 = NextSig(1); p3 = NextSig(1)
p4 = NextSig(1); p5 = NextSig(1)
p6 = NextSig(1); p7 = NextSig(1)
p8 = NextSig(1); p9 = NextSig(1)
pA = NextSig(1); pB = NextSig(1)

#----- Transport
K = NewKick()
K[p6] = [0,-1]      # All other signals
K[p2] = [0,-1]      # have [0,0]
K[p5] = [-1,0]      # by default
K[p9] = [0,-1]
K[pA] = [0,-1]

#----- Interaction
def shuffle():
    p6._ = p0;    p2._ = p4;    p9._ = p8;
    p4._ = p1;    p8._ = p5;    p1._ = p9;
    p0._ = p2;    p3._ = p6;    p7._ = pA;
    pB._ = p3;    p5._ = p7;    pA._ = pB

def swaps():
    if p4+pB+p9+p6==0: p0._ = p2; p2._ = p0
    if p8+p5+pA+p7==0: p1._ = p3; p3._ = p1

def event():
    swaps(); Latch(); shuffle()

MakeLookup(event)

#----- Step
def step():
    DoInteract(event)
    DoKick(K)

update_scan = step

#----- Init
def initialize():
    GetPattern("init.pat")
```

Figure 22: SIMP program for the polymer model implementation discussed above.

for example, `p4=NextSig(1)` requests the allocation of one more signal, to be called `p4`, consisting of a single bit. From now on, any reference to `p4` as a kick index in the Transport section will compile the appropriate layer *address* for the signal, while the appearance of `p4` in another context, namely as an r-value in expressions in the Interaction section, will force the lookup table generator to pass in review the possible *values* for that signal. In brief, signal names act as “smart” variables, acting as address or contents as appropriate to the context.

The Transport section specifies one or more “kick” vectors, that is, displacements by which the several layers are to be shifted during the transport stage; in our case, kick `K` embodies the right column of table (2). Different kicks may be appropriate to different phases of a complex macro-step.

In the Interaction section, the ultimate goal (`MakeLookup`) is to construct one or more lookup tables that specify interaction functions. In our case the lookup table we want to construct is called `event`. In turn, the logic structure of the process that `event` implements is given by the composition of `swaps` and `shuffle` (cf. Fig. 21). Normally, r-values like `p4` denote the state of a signal as of at the beginning of an interaction (the left-hand side of Fig. 21); the command `Latch` directs the following function component, `shuffle`, to take its inputs from the outputs of `swaps`, and thus in some sense embodies the dashed line of Fig. 11. Note that `swaps` performs swaps of central tile sites (cf. Fig. 14) like `p0` and `p2` conditional on the values of peripheral sites like `p4...p6`. On its part, `shuffle` implements the left column of table (2).

In the Step section, `update-scan` is a handle for the SIMP “control panel” (which may be a simple text-oriented affair or a fancy graphic user interface). When the user at the control panel “steps on the gas”, as it were, of the simulation, the “pedal” is connected to a box labeled `update-scan`. It is up to the user to fill this box with the desired contents—in this case a function object, called `step`, consisting of the interaction `event` followed by the transport operation (or “kick”) `K`.

Finally, in the Init section we detail how to specify the system’s initial state and any other initial settings, such as resetting a counter or seeding a random number generator. Here we ask to load a pattern file called `init.pat`.

For the moment, visual rendering of the programmable matter’s grid’s contents is handled by a collection of ad-hoc modules that implement appropriate defaults. Designing a general-purpose, programmable rendering engine to complement the programmable matter engine is an ambitious, open-ended task which we have barely started to address.

9 STEP implementation notes

The STEP API defines an abstract engine of great generality. For sake of efficiency, because of resource limitations, or for other reasons, a STEP implementation may not be willing or able to support the abstraction in its full generality; we may have an implementation that does at most two dimensions, or that limits the number of layers to 16, and so forth. For example, because of deliberate architectural choices, in CAM8 the length of each dimension must be a power of two (this approach is further explored in [21]). Such restrictions are quite legitimate in the STEP philosophy, provided that the user gets appropriate feedback through the API.

We shall not attempt to describe the API in detail here. Suffice it to say that provisions have been made for augmenting the interaction function with *extra input signals* (stimuli or “miracles”) supplied on a site-by-site basis from the outside world, and with *extra output signals* (“measurements”) that report some local properties of the system for the sake of the external world and are not “plowed back” into the mesh. There are also provisions for different kinds of boundary conditions (injection of signals at the open edges, wraparound options as discussed below).

Visual rendering of a programmable matter system, though not strictly part of the STEP functions, must find in the engine appropriate structured “hooks” to do perform its task well. For this reason, some rendering aspects are covered in the STEP abstraction.

If certain features are not used in a particular model or supported in a particular implementation, the implementation itself is encouraged to make ad hoc optimizations for what *is* used—for instance by compiling special, simpler and more efficient, versions of the most critical tasks.

On the other hand, we believe that it is useful for an abstract model to introduce well thought-out generalizations even if they are not likely to be frequently used, or they may make sense only for a software realization. We’ll offer one example below.

When one closes the rectangular grid of Fig. 8 onto itself to yield a torus, one must at the same time coherently close onto itself the coarser processor mesh of Fig. 15 which we had laid over the grid. This is possible only if the length of the torus along each dimension is a common multiple of both grid and mesh spacings along that dimension. Even then, if for modeling convenience one requires that the grid’s principal axes (or some other secondary axes, if preferred) be great circles of the torus, it may not be possible to have the processor mesh’s principal axes (which are “hard” features of the actual hardware or software) be themselves great circles. For this reason, before closing an n -dimensional parallelepipedal mesh of a certain size into a torus, STEP allows one—for each dimension—to shift the two faces to be glued with respect to one another along the remaining $n - 1$ dimensions (in this way, the mesh’s great circles along that dimension may become “great helices”, winding several times around the torus before closing onto themselves). This gives one much greater flexibility in choosing a discrete processor mesh, since it allows more kinds of mesh to coherently tile the target grid. It also gives greater flexibility in coherently rendering and viewing state data in grid, mesh, or display coordinates without distortion of angles.

With this in mind, to fully specify the toroidal geometry of the mesh in the general case, a dimension *vector* (a listing of the lengths of the n dimensions) is no longer sufficient, and one must supply an $n \times n$ *dimension tensor*. The STEP abstraction support this more general kind of toroidal mesh.

10 Conclusions

We have specified and are in the process of further developing a common language for conceptualizing and

implementing programmable-matter models. This language is accompanied by exemplar software engines that run on popular platforms and are remarkably simple and efficient.

Most of the existing cellular automata environments give preference to complex cells (tens or hundreds of bits), where evaluation of the transition function is by far the most demanding task and data transport and related infrastructural services are a mere afterthought. We tend to favor a finer-grained approach to modeling, with only a modest number of bits (say, ≤ 32) per site or event but a very large number of sites (millions to billions, for PC platforms). Here, data indexing and data movement represent a much larger fraction of the overall activity and it is important that they be efficiently managed.

A STEP engine uses the lattice-gas metaphor as a basic computational approach, whether for physics or for more mundane applications. Events and signals in such an engine may not bear a direct correspondence with the events and signals of the target system. The latter are synthesized from the former by processes (iteration, hierarchical build-up, etc.) that are quite familiar to the computing community even though they may not always be used in a traditional way. Much as in ordinary programming complex procedures are built from simple machine instructions, in a similar way complex materials or “states of matter” are built in programmable matter from elementary transport and interaction primitives—that is, elementary spacetime signals and events.

STEP, SIMP, and all that are available at <http://pm.bu.edu>, where we maintain a clearing house for programmable matter concepts, implementations, applications, and dissemination initiatives.

REFERENCES

- [1] ADORNI, G., S. CAGNONI, and M. MORDONINI, “Cellular-automata based optical flow computation for ‘just-in-time’ applications”, *10th Int. Conf. Image Analysis and Processing (ICIAP99)*, Sept. 1999, 612–617
- [2] BENIGNI, A., P. MENTRASTI, and T. TOFFOLI, “A model of urban transport simulation using the cellular machine CAM8”, M. DELORME et J. MAZOYER (eds.), Proc. of *Automata 99*, Workshop on Cellular Automata, 4th IFIP WG 1.5 Meeting, École Normale Supérieure, Lyon, France, 1999.
- [3] BENNETT, Charles, “Logical reversibility of computation,” *IBM J. Res. Develop.* **6** (1973), 525–532.
- [4] BENNETT, Charles, Norman MARGOLUS, and Tommaso TOFFOLI, “Bond-energy variables for Ising spin-glass dynamics,” *Phys. Rev. B* **37** (1988), 2254.
- [5] BURKS, Arthur (ed.), *Essays on Cellular Automata*, Univ. Ill. Press 1970.
- [6] DOOLEN, Gary, et al. (ed.), *Lattice-Gas Methods for Partial Differential Equations*, Addison-Wesley 1990.
- [7] FREDKIN, Edward, and Tommaso TOFFOLI, “Design principles for achieving high-performance submicron digital technologies,” from a proposal to DARPA, MIT Lab. for Comp. Sci., Nov. 1978.
- [8] FREDKIN, Edward, and Tommaso TOFFOLI, “Conservative Logic,” *Int. J. Theor. Phys.* **21** (1982), 219–253.
- [9] FRISCH, Uriel, Brosl HASSLACHER, and Yves POMEAU, “Lattice-gas automata for the Navier-Stokes equation,” *Phys. Rev. Lett.* **56** (1986), 1505–1508.
- [10] FRISCH, Uriel, et al., “Lattice gas hydrodynamics in two and three dimensions,” [6], 77–135.
- [11] GARDNER, Martin, “The Fantastic Combinations of John Conway’s New Solitaire Game ‘Life’,” *Sc. Am.* **223:4** (April 1970), 120–123.
- [12] HARDY, J., O. DE PAZZIS, and Yves POMEAU, “Molecular dynamics of a classical lattice gas: Transport properties and time correlation functions,” *Phys. Rev.* **A13** (1976), 1949–1960.
- [13] HASSLACHER, Brosl, “Discrete Fluids,” *Los Alamos Science*, Special Issue No. 15 (1987), 175–200 and 211–217.
- [14] HÉNON, M., “On the relation between lattice gases and cellular automata,” [24], 160–161.
- [15] JACOPINI, Giuseppe, and Giovanna SONTACCHI, “Reversible parallel computation: an evolving space model,” to appear in *Theor. Comp. Sci.* **75** (1990).
- [16] KARI, Jarkko, “Representation of reversible cellular automata with block permutations,” *Mathematical Systems Theory* **29** (1996), 47–61.
- [17] SMITH, Mark, “Representation of geometrical and topological quantities in cellular automata,” *Physica D* **45** (1990).
- [18] MARGOLUS, Norman “Physics-like models of computation,” *Physica D* **10** (1984), 81–95.
- [19] MARGOLUS, Norman, “Physics and Computation” (Ph. D. Thesis), *Tech. Rep. MIT/LCS/TR-415*, MIT Lab. for Comp. Sci., March 1988.
- [20] MARGOLUS, Norman, and Tommaso TOFFOLI, “Cellular Automata Machines,” [6], 219–248.
- [21] MARGOLUS, N., “Crystalline Computation,” in *Feynman and Computation* (A. HEY, ed.), Perseus Books, 1999.
- [22] MARQUES PEREIRA, R., G. ADORNI, and V. D’ANDREA, “A cooperating edge grammar for edge detection” in *Proc. 2nd Conf. on Cellular Automata for Research and Industry (ACRI’96)*, Springer-Verlag 1997, 158–167.
- [23] MCCARTHY, Wil, “Programmable matter”, *Nature* **407**(2000), 569.
- [24] MONACO, R. (ed.), *Discrete Kinetic Theory, Lattice Gas Dynamics, and Foundations of Hydrodynamics*, World Scientific 1989.

- [25] PIERINI, P., “Space-time structures for cellular automata”, S. DI GREGORIO and G. SPEZZANO (eds.), Procs. of ACRI '94, Rende di Cosenza, Italy 1994, 27-37
- [26] SMITH, Mark, “A Universal Source of Randomness for Digital Coins”, submitted to *Random Structures and Algorithms* and summarized in [36].
- [27] SMITH, Mark, Yaneer BAR-YAM, Y. RABIN, N. MARGOLUS, T. TOFFOLI, and C. H. BENNETT, “Cellular Automaton Simulation of Polymers,” *Complex Fluids* (D. WEITZ *et al.* ed.), Materials Research Society 1992, 483–488.
- [28] TALIA, Domenico, “Cellular automata processing for high-performance simulation”, *IEEE Computer* **33:9** (Sep. 2000), 44–51.
- [29] TOFFOLI, Tommaso, “Computation and construction universality of reversible cellular automata,” *J. Comp. Syst. Sci.* **15** (1977), 213–231.
- [30] TOFFOLI, Tommaso, “Cellular Automata Mechanics,” *Tech. Rep. 208*, Comp. Comm. Sci. Dept., The Univ. of Michigan 1977.
- [31] TOFFOLI, Tommaso, “Bicontinuous extension of reversible combinatorial functions,” *Math. Syst. Theory* **14** (1981), 13–23.
- [32] TOFFOLI, Tommaso, “CAM: A high-performance cellular-automaton machine,” *Physica D* **10** (1984), 195–204.
- [33] TOFFOLI, Tommaso, and Norman MARGOLUS, *Cellular Automata Machines—A New Environment for Modeling*, MIT Press 1987.
- [34] TOFFOLI, Tommaso, and Norman MARGOLUS, “Invertible Cellular Automata: A Review,” *Physica D* **45** (1990), 229–253.
- [35] TOFFOLI, Tommaso, and Norman MARGOLUS, “Programmable matter,” *Physica D* **47** (1991), 263–272.
- [36] TOFFOLI, Tommaso, “Programmable matter methods” *Future Generation Computer Systems* **16** (1999), 187–201.
- [37] TOFFOLI, Tommaso, “Fine-Grained Parallel Supercomputer”, Tech. Rep. PL-TR-95-2013, Phillips Laboratory, Hanscom AFB, Department of Defense, Nov. 1994, 197+viii pp.
- [38] TREDENNICK, Nick, “Technology and business: forces driving microprocessor evolution,” *Proc. IEEE* **83:12** (1995), 1641–1651.
- [39] ULAM, Stanislaw, “Random Processes and Transformations,” *Proc. Int. Congr. Mathem.* (held in 1950) **2** (1952), 264–275.
- [40] ULICHNEY, Robert, *Digital Halftoning*, MIT Press, 1987.
- [41] VICHNIAC, Gérard, “Simulating physics with cellular automata,” *Physica D* **10** (1984), 96–115.
- [42] VON NEUMANN, John, *Theory of Self-Reproducing Automata* (edited and completed by Arthur BURKS), Univ. of Illinois Press 1966.
- [43] VON ROSSUM, Guido, “Computer Programming for Everybody”, proposal to DARPA, Corporation for National Research Initiatives, July 1999, www.python.org/doc/essays/cp4e.html.
- [44] WOLFRAM, Stephen, “Computation Theory of Cellular Automata,” *Commun. Math. Phys.* **96** (1984), 15-57.
- [45] WOLFRAM, Stephen, “Statistical mechanics of cellular automata,” *Rev. Mod. Phys.* **55** (1983), 601.
- [46] WOLFRAM, Stephen, *Theory and Applications of Cellular Automata*, World Scientific 1986.
- [47] WORSCH, Thomas, “Simulation of cellular automata”, *Future Generation Computer Systems* **16** (1999), 157–170.
- [48] WORSCH, Thomas, “Programming environments for cellular automata”, liinwww.ira.uka.de/nworsch/, Lehrstuhl Informatik, Universität Karlsruhe 1997.
- [49] ZUSE, Konrad, *Rechnender Raum*, Vieweg, Braunschweig (1969); translated as “Calculating Space,” *Tech. Transl. AZT-70-164-GEMIT*, MIT Project MAC, 1970.