METHODOLOGY AND IMPLEMENTATION OF A SOFTWARE ARCHITECTURE FOR CELLULAR AND LATTICE-GAS AUTOMATA PROGRAMMING

TED BACH

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

BOSTON

UNIVERSITY

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

METHODOLOGY AND IMPLEMENTATION OF A SOFTWARE ARCHITECTURE FOR CELLULAR AND LATTICE-GAS AUTOMATA PROGRAMMING

by

TED BACH

B.S., Ohio University, 1999
M.S., Boston University, 2002

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2007

Approved by

First Reader

Tommaso Toffoli, PhD
Associate Professor of Electrical and Computer Engineering

Second Reader

Martin Herbordt, PhD
Associate Professor of Electrical and Computer Engineering

Third Reader

Roscoe Giles, PhD
Professor of Electrical and Computer Engineering

Fourth Reader

Lev Levitin, PhD
Professor of Electrical and Computer Engineering

Fifth Reader

Norman Margolus, PhD
Chief Scientist, Permabit Inc.

Acknowledgments

First, for supporting this work, I would like to acknowledge the Department of Energy who funded the first four years of my work at BU under grant number 4097-5. I would also like to thank the BU Department of Electrical and Computer Engineering for tuition support in my last year and a half and the Ab Initio Software Corporation for allowing me to take liberal amounts of time off as I finished writing and defending this thesis.

I have had the benefit of many kind and generous mentors and teachers I would like to thank. I'll start with my parents, Jim Bach, Nancy Bass, Barbara Rostad, and Mark Bass, who have supported me, shown me the wonder of the world and encouraged me to explore it. In my early education I am grateful to Mrs Stumpf who believed in me and helped me overcome early academic difficulties, and Mrs de'Lavalle who set me firmly on the path of academic success. Later, Kurt Nostrant introduced me to hands-on science in the "Science Olympiad" competitions. In high-school Sally Ball taught me the wonders of biology and Gary Dunfee the joys of the hard sciences of chemistry and science, and Stephanie Knight for literature, and Cynthia Kaldis for Latin and the classics.

At Ohio University, I would like to thank Mehmet Celenk for discrete math, David Bambeck for microprocessors, David Chelberg for algorithms. Additional thanks go to Charles Alexander, Dennis Irwin, Doug Lawrence, and Jim Watson for teaching me how to network and lead. At Sunpower Inc. and Global Cooling Manufacturing, a special thanks goes to David M. Berchowitz who gave me my first chance to cut my programming and modeling teeth and introduced me to the literature of complex systems. I would also like to thank Gary Wood, Doug Keiter, and Neil Lane for teaching me about real-world engineering.

During my years at BU I have had the pleasure of interacting with many excellent faculty. In addition to the members of my committee—Tom Toffoli, Martin Herbordt, Roscoe Giles, Lev Levitin, and Norm Margolus—I would also like to especially acknowledge Ari Trachtenberg, Alexander Taubin, David Castiñon, Richard West, and Jeff Carruthers.

Among the fondest memories of my time at BU are the paper discussion groups and seminars held at various times on topics ranging from artificial intelligence, to non-conventional computing, to quantum mechanics, to small-world networks. I would like to thank Jon Polimeni, Brian Rossa, Lee Lichtenstein, Lan Hu, Rob Pitts, Leo Grady, Kate Mullen, Suuriniemi Saku, Saikat Ray, Sachin Agarwal, and Silvio Capobianco for making these sessions so enjoyable.

Above all, I would like to thank Tom Toffoli for taking a chance on a young man from Ohio nearly seven years ago and providing an ever-open office door and endless thoughtful advice, impromtu mini-lectures, direction, and heaps of uniquely visionary food for thought.

Special thanks go to my lovely, briliant wife Kristi for her kindness and support and to my darling daughter Eleanor for pardoning my absence as I finished my dissertation. I love you both dearly.

METHODOLOGY AND IMPLEMENTATION OF A SOFTWARE ARCHITECTURE FOR CELLULAR AND LATTICE-GAS AUTOMATA PROGRAMMING

(Order No.)

TED BACH

Boston University, College of Engineering, 2007

Major Professor: Tommaso Toffoli, PhD, Associate Professor of Electrical and Computer Engineering

ABSTRACT

Cellular automata (CA) are a class of fine-grained, massively-parallel computational paradigms for describing discrete, spatially distributed dynamical systems governed by a discrete, homogeneous, local dynamics. CA computing applications include discrete physical models and empirical studies in algorithmic self-assembly, computation, complexity, and the statistical mechanics of emergence.

Despite a number of existing CA applications, programming environments, and special-purpose hardware and software techniques for implementing them, there exists no implementation-independent architecture for computing with CA that is analogous to those used in similar types of co-processing applications such as computer graphics. It is our thesis that, by analyzing the recurring high-level conceptual constructs and methods employed by the human designers of CA systems and the various low-level implementation strategies employed to achieve high performance in running them, one may arrive at a rational software architecture that makes these dynamical systems natural to express yet amenable to efficient, automated implementation.

We motivate and present a software architecture called STEP, short for space-time event processor, that targets these goals. The STEP architecture consists of an abstract CA co-processor, an application program interface for controlling it, and a programming environment called SIMP for building CA applications. We show that SIMP applications are considerably shorter and simpler than corresponding C programs, but, nevertheless, achieve performance equal to that of compiled C.

Novel aspects of this work include (a) the first realization of a simple conceptual and computing environment that can deal equally well with CA and variants such as lattice gas automata and partitioning cellular automata; (b) direct support for CA state variables and updates defined on non-orthonormal integer lattices; (c) a new method performing CA updates via a scan loop that efficiently processes the uniform part of the CA volume and employs a virtual-interrupt based interpreter to handle exceptions at the boundaries; and (d) techniques for converting and compiling Python CA transition function code into C procedures on-the-fly.

Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
BPCA	Block Partitioning Cellular Automata
CA	Cellular Automata
FHP	The “Frisch, Hasslacher, Pomeau” lattice gas
HNF	Hermite Normal Form
HPP	The “Hardy, De Pazzis, Pomeau” lattice gas
LUT	Lookup Table
LGA	Lattice Gas Automata
PC	Personal Computer
PCA	Partitioning Cellular Automata
SIMP	Simple Interface to Matter Programming
STEP	Space-Time Event Processor

Chapter 1

Introduction

Complex systems often arise as the macroscopic aggregate of simple, local, spatially-distributed, microscopic interactions. Cellular automata (CA) and lattice gas automata (LGA) are a mathematical and computational formalism for describing such systems. Due to their generally non-linear nature and computational accessibility, CA and LGA systems are often used to study empirically dynamics where analytic results are difficult to obtain.

The canonical example is turbulent fluid flow. Fig. 1·1 depicts a 2-dimensional turbulent fluid flow past an obstacle. This "wind tunnel" is implemented as a LGA. Both the fluid flow and the white tracing material are implemented with a fine-grained CA/LGA dynamics. Bits represent particles and both the macroscopic fluid flow and its effect on the tracer material arise from particles that move and interact locally and in a homogeneous fashion.

Structurally, CA may be thought of as a uniform, spatially extended network of simple processing elements that communicate via local signals and compute local update 'events' as a function of them. M.C. Escher's "Cubic space filling" in Fig. 1·2 (a) is an appealing 3-dimensional depiction of such a network.

Fig. 1·2 (b) shows the result of iterating a "majority" dynamics in a CA network that has the same structure as Escher's drawing. In this dynamics, each site has a single bit of state that is updated locally as function of itself and its neighbors. The event processing nodes map the next state to the majority value if the site itself

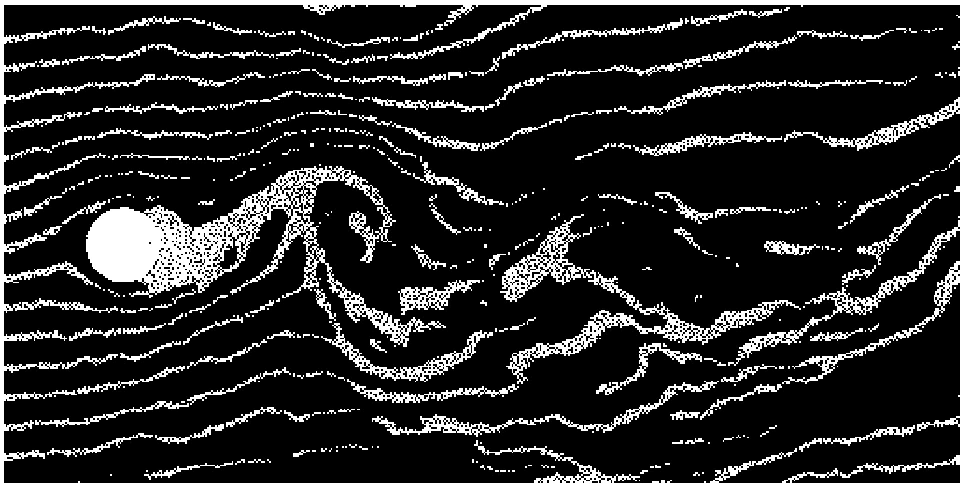
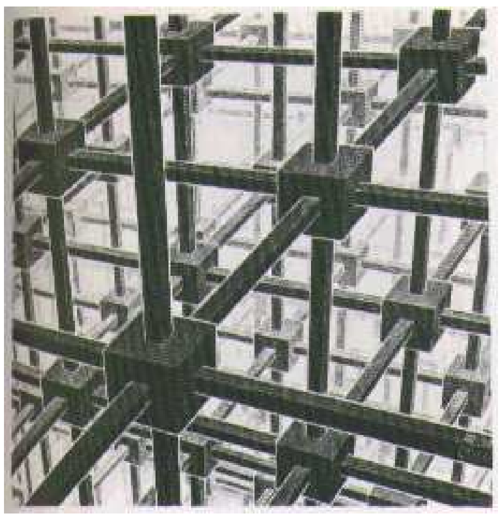
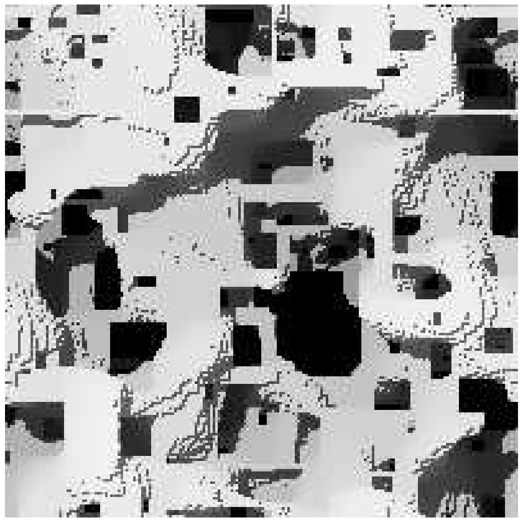


Figure 1·1: Simulated FHP lattice gas wind tunnel

and that of its six nearest neighbors. Iterating this local dynamics leads to a sort of "programmable matter" in which isolated clusters of particles are prone to evaporate and unform domains form. In the figure, bits with a value of 1 are rendered as a solid white material while bits with a value of 0 are transparent. (The image is the result of iterating 500 parallel majority votes on an initally random distribution of states.)



(a) "Cubic Space Filling"[1]



(b) 3D majority

Figure 1·2: Spatially distributed discrete computing

1.1 CA applications

Because they are a discrete structure with much similarity to physics, CA are often employed to create physical models. CA were invented by Ulam and von Neumann (von Neumann, 1966) as an *ad hoc* paradigm for studying reproducing machines. If they had not invented the concept, they would most likely have been invented not long thereafter (perhaps with an easier-to-pronounce name). The reason is that CA are a relatively natural discrete counterpart to differential equations. They provide a mechanics for describing *discrete* distributed systems and, as such, are a tool of general utility.

With the advent of fast, powerful, cheap computing, CA have gone from being an interesting curiosity *á la* Conway's Game of Life to a practical tool for simulating a variety spatially-distributed, discrete systems.

Because simulating a large CA volume is inherently compute-intensive, the paradigm is most often used for modeling systems whose dynamics are complex and defy simpler description. Examples include a host of fluid dynamics applications (U. Frisch and Pomeau, 1986; Hénon, 1987; Harris, 2000). CA have also been used to describe various types of flows as well as non-homogeneous flows and biochemical simulations (Weimar, 2002). Novel models of collision-based computing and various mathematical studies of pattern formation such as models of crystal formation have also been carried out and continue to be developed (Harris, 2000).Other applications range from biological processes and drug reaction modeling, to pedestrian traffic, to pattern recognition and to quantum computing (Bandini et al., 2002). We will not recount all the applications here, suffice it to say that there are many and the applications are discussed in a number of places (Toffoli and Margolus, 1987; Weimar, 1997; Ilachinski, 2001).

Aside from constructing physical models, CA also provide a model of fine-grained massively parallel computing, a sort of ‘calculating space’ (Zuse, 1969), By the same token that locally connected, spatially distributed structure of CA are suitable for simulating large spaces governed by a uniform microscopic physics, CA computation is ideally suited for fine-grained massively parallel implementation. One could well imagine that the crystalline structure of CA computations might, some day, actually be mapped onto real crystals or some other physical substrate. Indeed, quantum dot CA (Imre et al., 2006) represent a current step in this direction. In addition, Winfree, Cook et al. have been able to create DNA crystals that, in the process of their formation, carry out a one-dimensional CA dynamics (Cook et al., 2004; Schulman et al., 2004) (Winfree and Bekbolatov, 2004) (Cook et al., 2004). The goal is to create self-assembling circuits.

Cellular automata are often employed as a vehicle for studying complex systems in the abstract (Wolfram, 2002; Griffeath and Moore, 2003) and information mechanics (Toffoli and Margolus, 1990; Toffoli, 1999; Griffeath and Moore, 2003). Research also continues in the study of CA as a substrate for self-reproducing machines (Mange et al., 2004).

1.2 CA programming environments

In essence, specifying a CA is quite simple. The state of a CA or LGA is defined on a discrete grid (regular lattice) composed of a large number identical state variables, and evolves in discrete time steps according to a simple, local dynamics. It consists of a space, parallel state variables that fill the space, and a space-time iterated local dynamics for updating them.

The simplicity and uniformity of CA models makes them relatively easy to implement on a personal computer. A web search for a popular CA dynamics like Conway’s Game of Life yields hundreds of implementations. However, most are *ad* *hoc*, supporting only one dynamics or a small parameterized family of them and providing only rudimentary, canned facilities for auxiliary tasks such as rendering, gathering statistics, and initializing data.

The researcher—as opposed to the hobbyist towards whom most CA software is directed—requires flexible tools that endow her with the freedom to define her own CA experiments. And, although the `C` programming language is such a tool, she would rather concentrate on conceptual issues than be burdened with low-level issues—optimization, visualization, boundary condition handling, *et cetera*—that writing an implementation from scratch would entail. Instead, she will desire a CA programming environment that abstracts away accidental details and leaves her the time to focus on high-level programming and modeling aspects.

Such environments exist, but they vary in their simplicity, efficiency, flexibility, availability, and portability. For example, NetLogo (Wilensky, 1999) has a relatively simple programming environment and is flexible in that it can handle a wide variety of distributed systems other than CA; however, it is not efficient. Mathematica is flexible (Wolfram, 2002), however, it is not simple, freely available. Furthermore, it is not really implementation independent. Of the many environments—note that we make no attempt to provide a comprehensive survey of them here—see (Worsch, 1996) and (Talia, 2000) instead—JCASim (Freiwald and Weimar, 2002) probably comes the closest to the balance of simplicity, efficiency, flexibility, availability, and portability that we seek; however, its Java-based implementation and syntax, currently, are not entirely efficient or simple.

1.3 Our thesis

It is our thesis that by analyzing the recurring high-level conceptual constructs and methods employed by the human designers of CA computation systems and the var-

ious low-level implementation strategies employed to achieve high performance in running them, one may arrive at a software architecture that makes these systems natural to express yet amenable to efficient, automated implementation.

In this document we motivate and present a software architecture that targets these goals. The architecture is called the STEP architecture and it has two parts: SIMP, which provides the designer with a high-level scripting environment, and STEP, which provides an abstract interface to machine implementations. By way of example we introduce the conceptual and practical issues related to programming cellular automata, lattice gas, and block partitioning cellular automata systems and explain the approach we take to addressing them in SIMP. We discuss the STEP interface, the strategies employed in a fast PC implementation of it.

The architecture is built on collective past experience in programming and efficiently implementing CA experiments on various hardware and software platforms (Toffoli and Margolus, 1987; Margolus, 1994; Toffoli, 1994; Smith, 1994; Toffoli, 1999). We aim to externalize the abstractions and methods that have accrued in the CAM community and provide a balanced mix of simplicity, efficiency, generality, availability, and portability.

Although CA represent a data-parallel computational paradigm bearing much similarity to data-parallel, array-based computing represented in a wide variety of array programming languages including Matlab, APL, ZPL, and parallel varieties of Fortran, these languages general purpose languages. They are not specially tuned to the issues connected with programming cellular automata. The STEP architecture is distinct from such efforts in that it is a specially tuned, domain-specific (Spinellis, 2001; Mernik et al., 2005) programming and implementation architecture. By virtue of addressing a limited domain the architecture is afforded the opportunity to make optimizations that a more general-purpose program can not afford to make.

In the context of this architecture, we define some constructs that aren't available in any of these other programming environments. In particular, also introduce the notion of state variables that can be defined and directly and simply programmed on sublattices of the orthogonal integer grid. This can help make CA easier to program by preserving the original coordinates of the problem. It also enables one to program LGA and block-partitioning cellular automata (BPCA) in a way that is more natural than would otherwise be required (see Section 8.1.3 for more details). The LGA and BPCA families of CA programs are interesting because, as Section 3.5 discusses, unlike ordinary CA they can trivially be made to preserve local quantities, a property that is quite useful in designing "programmable mattter" applications.

We show that SIMP programs are considerably shorter and simpler than corresponding `C` programs, but, nevertheless, can achieve performance close to that of a compiled program. Because the STEP interface is generalized, it is theoretically extensible to a wide variety of hardware and software implementations. We outline the possibilities in this area and future directions for extending the functionality of the SIMP programming environment.

1.4 The STEP architecture

The STEP architecture represents an effort to repeat for CA computing the common hardware/software architectural pattern in which a generic interface is developed for a reasonably well-understood type of computing resource. The point is to decouple the specific implementation of a resource from the programs that use it. This affords an architecture the flexibilitly to develop implementations and programs/programming environments independently. It makes clear the capabilities of the architecture and the responsiblities of the implementations. It allows an application to select the best/most optimized implementation available. It enables cross-validation strategies by which one may compare a more unstable, but faster optimized implementation to a slow, but trustworthy reference implementation.

Of the many examples of this pattern, which includes such diverse instances as the Intel x86 instruction set architecture, the POSIX system services, the Java virtual machine, and the SQL interface to relational databases, the one most closely akin with the requirements of CA-based computing is the OpenGl interface to the resources of an abstract computer graphics card. Although there are a number of CA programming environments, none seem to be based on an abstract CA processor and our work is novel in this regard.

As Fig. 1·3 depicts, our STEP architecture consists of a CA processor, a CA programming environment, and CA applications. In particular, we formulate an abstract notion of a CA processor called a STEP (space-time event processor) together with an application programmer's interface (API), and develop a programming environment called SIMP that builds on the STEP API. In order to flesh out the architecture we develop fast, efficient PC STEP implementations and construct CA application programs that demonstrate how to use SIMP.

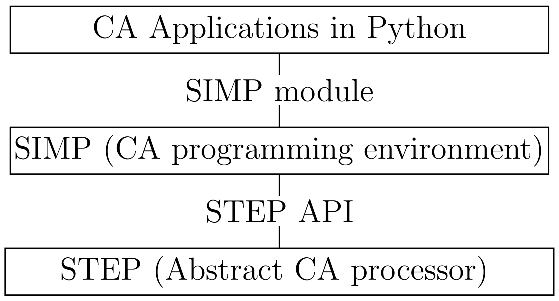

CA Applications in Python
SIMP module
SIMP (CA programming environment)
STEP API
STEP (Abstract CA processor)

Figure 1·3: The STEP architecture

Our work originated as a CA programming environment based upon the machine-dependent STEP Forth programming environment of the CAM-8 cellular automata machine. The STEP API remains loosely based upon this earlier model.

By no means do we consider our STEP architecture to be the last word in this area. Rather, our goal is to provide a working 'blueprint' for this kind of approach to CA computing by drawing out the issues at play and constructing an exemplar architecture and implementations that serve as a proof of concept and demonstrates some concrete results. We intend for this blueprint to be clear enough to guide the way for the further development of the STEP architecture or to be incorporated in part or whole in some future architecture.

For reasons of efficiency and implementability, the STEP platform is limited to CA and related applications having a uniform, crystalline space-time structure.

As Snyder eloquently argues (Snyder, 1986), there can be no single model for parallel computation. Unlike the variations among serial von Neumann computer architectures, the variations among parallel computer architectures are great enough that ignoring them results in significant performance degradations. Because the best a parallel computer can do is to achieve a speedup equal to the number of processors (Amdahl's Law), any degradation due to 'friction' between the programmer model and its implementation limits the already modest speedup factor. Therefore, the nature of the architecture and the costs of various operations must be exposed to the programmer and no single model can be used for all architectures.

Compilers can do a much better job of mapping applications to hardware when the domain of discourse is limited. For example, technology mapping a general-purpose program onto an FPGA is hard; however, compiling STEP CA computation primitives would, by virtue of regularity, be easier.

Because the goal of an all-encompassing model is not tenable, we seek to carve out a model that is good from both the programmer's perspective and from the perspective of implementability. Doing so requires balancing the desire to widen the application domain versus the need to maintain efficiency. Therefore, this work limits STEP's scope to architectures that support CA, LG, and PCA efficiently and would only expand it in ways that do not penalize these primary applications.

Due to the simplicity and uniformity of the paradigm, CA can be implemented very efficiently on a wide variety of computer architectures, allowing the experimenter to deploy the large space-time swaths necessary for studying complex macroscopic phenomena. However, implementation strategies themselves are not always simple or trivial, especially when one wants to achieve high performance or do computing on structures that don't map directly into computer memory. But, an experimenter cares not how a CA is actually implemented, so long as it is fast and correct, and would generally rather spend time on things like defining and refining models, adjusting parameters, and running experiments than on doing low-level, implementation-specific programming.

In the present work, we choose to focus on defining an architecture for computing with CA that allows one to run efficiently on an ordinary computer, and in effect turns every computer into a potential CA laboratory.

We do not make the implementation of fast CA hardware the focus of our work. It is already well known that "embarrassingly parallel" applications like CA map efficiently onto a variety of parallel computing resources under a variety techniques.

Moreover, a history of adventurous machine designs—such as (Margolus, 1994; Toffoli, 1984; Hillis, 1986; Ikenaga and Ogura, 2000)—has proved that pure speed and efficiency alone do not a successful machine make. To date, special-purpose hardware for CA computation has not gathered the critical mass—in terms of applications, developers, and relative demand—necessary to sustain the commercial production of dedicated hardware. This is not altogether surprising as general purpose computing enjoys a significant market subsidy by virtue of its broad applicability.

Rather than taking an "If you build it, they will come" approach, the present work takes more of an an "If they come, you will build it" approach. The thrust is on making solid high-level programming environments that enable rapid prototyping using any computational resources available; the understanding is that, should greater speed be needed, faster implementations can always be purchased. For the time being, it is more cost effective to hitch a ride on the common PC, which, owing to its heavy market subsidy, is the most cost effective solution despite the fact that the same components that make up the PC could be rearranged into a much faster engine dedicated to CA computation. Therefore, in this work, we focus on fast PC implementations.

But, our main focus is on the development of high-level programming environments and constructs that consolidate existing knowledge (Toffoli and Margolus, 1987; Smith, 1994; Resnick, 1994; Weimar, 1997; Toffoli, 1999; Wolfram, 2002) about representing CA and LG. Ideally, a programming environment should *structure knowledge*, making programs simpler to write so that programmers, in turn, can develop more complex applications.

Our aim is to have an application development architecture designed in a way that allows one to simply "plug in" the appropriate computing resources when they are available. The STEP API does this by simply communicating its expectations to the back-end that implements computation without forcing the implementation to resolve too many front-end programming issues. In order to make the interface efficient, we take the general properties of variety of implementation strategies into account (see Chapter 3.6) in defining our architecture for computing with CA.

Chapter 2

Overview and Summary

We want to people to be able to program CA in a way reflects the simplicity, elegance, and computational parsimony of the paradigm. We want CA programs that are high-level from both a programming language and conceptual standpoint, and, yet, efficient from an implementation standpoint.

To target these goals, we advocate the development of a domain-specific architecture for computing with CA. We posit that such a architecture requires: (a) *a mature, high-level programming language* in which to write applications; (b) *a set of rational, implementation-independent constructs for expressing CA computations*; and (c) *a collection strategies for efficiently implementing CA on ordinary computers.*

This thesis motivates and proposes a novel architecture for CA computing, describes how we implemented it, and demonstrates its efficacy in the writing of high-level yet efficient CA applications. In the course of developing this architecture, we've made a number of contributions to the state of the art, including

- A CA programming and computing environment whose applications achieve performance comparable to that of hand-coded C in a fraction of the programmer time and with a fraction of the number of lines of code. (Chapter 8)

- An interrupt-based CA update implementation method that runs a tight, uniform loop in most of the volume, but, at minimal cost, can generate virtual interrupts at pre-computed locations to handle non-uniformities such as the boundaries. Because the non-uniformities scale with the surface and not with the volume, the cost of handling the interrupts is amortized by the speed of updating the volume. (Chapter 7.2)

- A just-in-time C-code generator that, depending on the complexity of a transition function, can most efficiently implement it as either a lookup-table or a compiled C procedure. (Chapter 7.4)
- A simple threaded, data-parallel implementation that achieves near ideal parallelization utilizing up to 8 CPUs on a shared-memory multi-processor computer. (Chapter 7.3)
- A software engineering blueprint for future CA implementations, programming environments, and architectures. (Chapter 5)
- A design infrastructure that combines lattice-gas shifts and CA neighbors in a single programming environment. (Chapter 4.4, Chapter 3.2)
- An original programming and implementation formalism in which state variables and dynamics can be defined on non-orthonormal integer lattices. This, for the first time, allows the developer to directly program in the block-partitioning cellular automata paradigm. (Chapter 4.5, Chapter 4.5)

This rest of this section goes into more depth, presenting a summary of our work and highlighting our contributions in more detail. In doing so, it recapitulates the overall structure of our thesis in the following areas

- the STEP architecture, a general architecture for CA computation,
- SIMP, a practical CA programming environment with examples that demonstrate its efficacy,

- the STEP API, our proposed interface to an abstract CA co-processor,
- efficient CPU-based STEP implementations, and
- novel constructs for programming CA on non-orthonormal integer lattices.

2.1 The SIMP programming environment

The whole point of having an architecture for CA computing is to create a high level programming environment. To demonstrate the utility of the architecture, we spend some time demonstrating useful CA programming constructs for writing CA, LGA and BPCA applications in the SIMP programming environment. The aim of SIMP is to make CA natural to program and provide the programmer with constructs that contain implicit knowledge about how to write CA applications.

In its implementation, the SIMP is a Python module that we've tuned for writing CA applications. To use it, a Python program just imports the SIMP module. The SIMP module both exposes the STEP API and builds extra functionality not included directly in the API in order to create a Python programming environment for cellular automata. SIMP owes much of its power and versatility to the fact that it is embedded in Python. In combining the STEP operations with the power of the Python programming language SIMP empowers the CA programmer to express programs at a high level with the full aid of the built-in Python libraries and extensive third-party modules. (Libraries for plotting data, manipulating images, and performing statistical analysis are among a few of the options that a CA application programmer might find useful.)

We have attempted to embed a significant amount of knowledge about programming CA into SIMP. The result is pleasing in that SIMP applications are fairly easy to write, taking much less time than lower-level programs. SIMP code is much shorter than similar code written in C. It is also comparable in size to programs written in other high-level CA environments such as Net-Logo, whose implementations are orders of magnitude slower. We demonstrate the utility of SIMP through tutorial-like examples.

In addition to our own work with SIMP, other colleagues and researchers have already employed it to perform CA-based studies. With the recent addition of the STEP extensions for sub-lattices, BPCA, and enhanced types that support mixing binary, integer, and floating-point computations, we expect that this user-base will continue to grow.

In most CA, the state variables and update events defined on the rectangular *grid* of an orthonormal integer lattice—that is a lattice whose sites are at a unit spacing in each dimension. This is a natural first choice in defining a CA, and maps directly into computer memory. However, in the context of some CA, many LGA, and all BPCA dynamics, the problem coordinates are not necessarily those of a square grid, but rather some other lattice with different generator vectors.

LGA and BPCA are particularly important forms of CA because, unlike n-input, single output CA, the n-input, m output form of their local functions can be used to create a local dynamics that, by virtue of conserving local quantities (like particle counts, energy and momentum) can actually preserve them globally as well. They present an ideal way of programming local quantities that achieve a global effect.

One example is the HPP lattice gas (Hardy et al., 1976), in which the dynamics is defined by four binary "particle bearing" state variables that propagate up, down, left, and right on a 2-dimensional grid and interact at the sites of the grid in local momentum-preserving collisions. Because of the way a particle moves (up, down, left, and right) it will alternate between odd and even sites of the grid; particles that start out on an odd site never interact with those that start out on an even site.

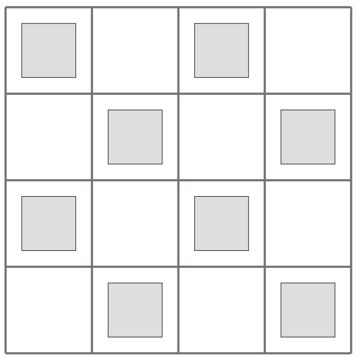


Figure 2·1: Checkerboard sublattice of the grid

In situations like these, there's really no reason to to simulate two interlaced but non-interacting dynamics. Instead, it's desirable to specify the dynamics in a way that includes just one of the two sub-grids. Ideally, one would program a dynamics like HPP by allocating its state variables on the even sites of the grid, shift them up, down, left and right and apply the interaction rule only at the alternating odd/even subgrid sites where they land. With the grid sublattice extensions to the STEP API one can do exactly this.

As far as we know, no other programming environment supports programming in this way, although some functional array languages (Chamberlain et al., 1999b; Chamberlain et al., 1998) seem to come close that apply at various stride spacings in an array.

Because sublattice constructs are not usually available in programming environments, CA on sublattices are normally programmed by packing state variables into an an orthonormal grid. This can be done by laboriously altering the geometry of the original problem and re-encoding the dynamics in a way that either requires two temporal or spatial phases[1] or requires adjusting the inertial frame of the problem (Bach and Toffoli, 2003). In addition to making the programming task more difficult, these re-adjustments can alter the geometry in ways that makes reading and writing state variables more difficult and skews the global geometry.

The sublattice constructs are also necessary for programming block partitioning cellular automata (BPCA). In BPCA, blocks of sites are updated simultaneously. By defining events that perform local state transition on sublattices, one can control the blocks of sites updated together as well as how the blocks-partitioning shifts between updates.

Although CA, LGA, and BPCA are all equivalent (they can be expressed in terms of one-another), and it is generally possible to re-express problems that geometrically inhabit lattices other than a square grid, doing so generally alters the problem space in a way that makes programming the dynamics and reading, writing, and rendering state variables much more difficult. Therefore, it is significant that SIMP and the underlying STEP API include ways to express computations in each of these three different ways and let the implementation decide how best to implement each rather than forcing the programmer to use one specific paradigm.

To support these constructs, a STEP must must learn some new tricks. In particular, a STEP needs to have a strategy for storing state variables on sublattices and performing I/O with respect to them, and posses CA event implementation logic that can handle sublattices. We discuss some of the possibilities and our approach towards addressing these issues in our STEP implementations. Incidentally, these issues are a generalization of what one used to have to do manually to program LGA and BPCA.

To support this, we extend the STEP API to include facilities for declaring state variables and events on arbitrary integer sublattices of the grid. We also extend the LGA shift operator so that it can be used to shift state variables and update events

[1]A common technique is the "Margolus Neighborhood" based recoding of HPP as a BPCA and then embedding the BPCA in an ordinary CA.

to the different *cosets* that their sublattices induce with respect to the grid.

We demonstrate the utility of the abstractions for non-orthonormal lattices by presenting the HPP lattice gas programmed on a sublattice and a model of polymer dynamics programmed as a block partitioning CA. We show that these approaches result in a significant reduction in the amount of SIMP code and make it possible to specify simple rendering rules that preserve the geometry.

In BPCA, the local CA update event applies to state variables in non-overlapping, space-filling blocks of sites. The blocks partition the space. Between updates, the block partitions shift. A common BPCA example is the Margolus Neighborhood for implementing the HPP lattice gas. It shifts between two non-overlapping sites and, in doing so, drags the state variables with it.

To program a block partitioning CA, some programming environments allow a CA transition function to parameterize what it does depending on the local coordinates of the site it is updating. Because, usually, CA transition functions are required to be translation-invariant, this is not always possible in a CA, so another technique is to program the dynamics so that it takes as an extra input, a constant state variable initialized with a pattern that indicates what coset a site is on (e.g., whether it is odd or even). The dynamics uses the coset information to parameterize its local behavior so that it effectively becomes block-partitioning.

We restricted lattices to the integers because allowing generic lattices would introduce a number of numerical and implementation problems that we chose to avoid in the first cut. Perhaps a meaningful semantics could be found for this later, but in the mean time, having integer sublattices is still fairly flexible even for representing arbitrary rational lattices—because, by scaling a lattice, one can find an arbitrarily close approximation in an integer lattice.

2.2 The STEP API

Our notion of the role of a CA processor is similar to that of a video card, *a STEP is more of a co-processor than full-fledged processor in its own right.* In particular we have chosen to base our architecture on a model that, like the video-card/api/application model of OpenGl, provides a means for declaring and issuing operations, but relegates control decisions to a controlling program run on a general-purpose processor.

Based upon a canonical collection of CA applications and a variety implementation strategies, we propose an abstract CA co-processor called a STEP (space-time event processor) and an API by which an application may initialize a STEP with a spatial extent, allocate typed, parallel state variables, instantiate parameterized instances various types of CA operations, issue them, and perform I/O.

For the first time, the STEP API gives the programmer access both CA and LGA programming constructs within the same environment. The STEP API provides both an *event* operation for defining CA update events via a local transition function and a *shift* operator for shifting data to neighboring sites. (Existing environments provided either the ability to read data from a neighbor site (CA) or to shift data between sites (LGA).) Other operations include read and write operations and operations that help with state variables defined on sublattices of the grid.

At initialization time, one specifies the name of the STEP implementation and the dimensions of the rectangular, toroidal space in which CA state variables will be defined. We concentrate on periodic boundaries because they are simple to describe and permit one to implement other types of boundaries by programming them within the dynamics.

Because a STEP requires the freedom to store state variables in any way that it finds fit, or may store the state variables off machine, the actual storage of state variables is entirely hidden. State variables must be accessed through special read and write API operations

The STEP API manages the allocation of state variables and defines a number of types for state variables. These types include binary unsigned integers with 1 through 7 bits; 8-, 16-, and 32-bit signed or unsigned integers; and 32-bit floating-point numbers.

Because implementations can, in performing optimizations and pre-compiling code, take significant advantage of knowing ahead-of-time as much as possible of the program to be executed, STEP includes a “Sequence” primitive for batching up operations. The STEP API features an asynchronous I/O protocol that permits the programming environment to keep a STEP running constantly without having to block for reads and writes. Even I/O operations can be embedded in sequences. This is implemented via callback functions that can be inserted into a sequence and employed to adjust the input and output buffers of read and write operations. If blocking operations are required, an implementation can, nevertheless, issue a blocking “flush” operation that forces the STEP to finish all pending operations or insert callback methods in order to get notification when a pariticular I/O operation has finished and its buffers are ready.

Because an implementation may not be able to support the full generality of the STEP API, we make the API permissive. If an implementation can not perform a certain type of operation, it reports this at the time that the operation is declared.

Because we are concentrating on applications layer in the Python programming language, our concrete implementation of the STEP API is in Python, but, should it become necessary, we could recast the API in a lower-level language like C or C++.

2.3 SIMP implementation

SIMP employs Python functions to support expressing an event in a way that is simple and natural in the context of a Python program. Based on this Python function, it generates a lower-level STEP representation of the event that is suitable for implementations. In particular, it handles the setting of default values, the specification of inputs and outputs of the local event and their "neighbor" offsets. Based upon the high-level transition function, it generates a low level version of the event that includes an explicit 'port binding' that lists the inputs and outputs of the local transition function and expresses the function in terms of these types.

SIMP must analyze and validate the function and transform it into a form that is suitable for a STEP to implement. This includes

- compiling a 'port description' for the event's local inputs and outputs and neighbor offsets,
- transforming the transition function to a form that has explicit input parameters and output return values,
- parameterizing the transition function by substituting global variables with the current values referenced in their scope,
- enforcing constraints on the function such as prohibiting the access of "global" values.

2.4 STEP implementations

We introduce some STEP implementations that efficiently run CA on general purpose CPU's and more advanced hardware such as multi-CPU shared memory computers.

In particular, these implementations have performance that is on-par with hand-coded C implementations. On a modest laptop computer (single-CPU, 1.5 GHz Pentium 4) they easily achieve rates in the tens of millions of sites per second for the CA examples that we present.

2.4.1 LUT/C transition function code generator

Many CA implementations take advantage of the fact that a CA transition function typically operates on a small number of inputs and can thus be compiled into a lookup-table (LUT) and implemented with a single memory lookup. Because the size of the LUT grows exponentially with the number of bits of input to the transition function, this optimization is only fast or practical when there are not too many bits of input state.

To combat this exponential growth, we have constructed a STEP implementation that can compile transition functions into C code and switch between a LUT and C procedures on-the-fly when the size of the inputs exceeds a default threshold of 16 bits[2].

Because the code generator supports the conversion of CA transition functions to compiled C code, it can support CA with transition functions having arbitrary numbers of bits of inputs including lattice Boltzmann automata with integer types, and continuous CA with floating-point types. The code generator works in concert with the STEP API's dual representation of a transition function as both a Python function and an abstract syntax tree.

[2]Because the cutoff may vary from machine to machine, we include a mechanism by which one may parameterize the cross-over point for a given implementation

2.4.2 Interrupt-based CA scan

We employ a novel method of implementing CA update loops with 'virtual interrupts' to handle irregularities in the computation, such as boundary conditions. Typically, these irregularities scale with the surface, so even if the cost of handling them is larger than for a typical site, it will be amortized so long as the rest of the volume is updated efficiently. In the method we introduce, the volume scan incurs only the cost of maintaining and checking a single interrupt counter, yet can handle an arbitrary number of interrupts.

The implementation scheme involves creating a scan loop that primarily just applies the transition function and increments pointers used to access input and output data, but also, at each iteration, decrements an *interrupt counter*. When the counter reaches zero, the loop drops into an a mini-interpreter that does things like modifying pointers to implement wrap-around within a row, changing the pointers of the scan to the next row, and exiting the scan routine altogether. Because the interrupts are infrequent, the cost of entering this interpreter is minimal.

The scan itself is expressed as a scan program written in the bytecode of the scan interrupt interpreter. We detail methods for compiling this code. Basically, the approach we take is to compute the beginning offset of each pointer for each row of the scan, then compute the points at which they wrap around and adjust the pointers there. We employ this method to implement periodic boundaries without needing to perform a modulus computation. The method could just as well be adapted to a variety of other situations—other boundary types, special updates for certain sites, exceptions to the uniform topology—where the normal case is to increment but sometimes there are exceptions.

2.5 Quantitative Analysis

In the end, the STEP architecture should be evaluated on its ability to satisfy the stated goals of making CA easier to program and more accessible, and on how well the API works as an intermediary between the front-end programming environment and back-end implementations. It's one thing to have a set of definitions and constructs, another to combine them in a natural fashion. In the SIMP programming environment chapter, we demonstrate the constructs that SIMP and STEP provide in practical contexts that demonstrate their generality and coherent use in various applications.

If the STEP architecture makes both front-end programming environments and back-end implementations easy to use—in both theory and practice—then the STEP architecture will have been successful.

In the quantitative analysis chapter we show that SIMP programs are much simpler and faster to write than corresponding C programs, yet, nevertheless, achieve a nearly equal performance. We also show that SIMP CA programs compare favorably with the StarLogo high-level programming environment which is orders of magnitude slower.

Because we have separated implementation from program, we are able to compare a few different implementations on the same set of programs. We explore this in Chapter 8. We highlight the performance factors in programming STEP. Even though the STEP API is written in Python, the overhead is not so significant. To issue operations from SIMP, there is an overhead, but this is amortized for CA with reasonable size.

We also demonstrate the effectiveness of the C compiling implementation. By switching from a LUT to C code when the size of the inputs are larger than 16 bits, it allows one to efficiently transition to large state sets.

We also see the effect of the number of neighbors on the performance. Clearly, reading more state variables tends to degrade performance, but it does so in a roughly linear fashion.

We also show the effect of the size of the space on the performance. Larger sizes have better performance because they amortize the overhead of irregularities at the boundaries and amortize the overhead of the Python programming environment. Beyond a certain size, though, cache effects can impinge on performance. This suggests that a nice future implementation would be block partitioning for preserving locality.

Chapter 3

Survey of CA applications and implementations

In order to propose an architecture for computing with CA, one must take both applications and implementations into account. This chapter does exactly that. It provides an extended to introduction to and survey of useful CA programming constructs. It also highlights various implementation strategies. Along the way, we outline the implications in terms of the types of programming constructs the architecture must support and the general properties of the implementations to which it must provide a common, yet efficient, interface.

This work addresses practical matters relating to the programming of CA. Abstract theory is not our primary concern in this section. (Later in Chapter 5 we give a more formal definitions) Therefore, we present general concepts through practical, canonical examples.

These examples do not represent the most recent or novel CA applications; rather, they've been selected to help us convey the issues at play in writing and implementing CA applications. Although this thesis strives to be self-contained, it is beyond the scope of the present work to provide a complete introduction to or an overview of CA and related modeling techniques. For this, we instead refer the interested reader to works such as (Toffoli and Margolus, 1987; Weimar, 1997; Ilachinski, 2001).

We mostly present two-dimensional examples. They are both easier to explain and used more often than three and higher dimensional models. Nevertheless, the concepts presented and our proposed CA architecture generalizes to n-dimensional spaces.

Before continuing, we give a brief overview of the rest of this chapter. In Section 3.1 we introduce basic CA via an 'excitable medium' model. We extend this basic model to a stochastic version and outline a kind of 'percolation' experiment where one uses it to construct a simple model of forest fires and study phase transitions. Along the way, we see that programming a CA application involves

- declaring global topology,
- allocating parallel state variables (we call them *signals*) on the sites of an integer grid,
- specifying a local, uniform dynamics (we call it a CA *event*) that, at each site, simultaneously maps the current state to the next state according to some local function of the values of neighbor sites,
- having techniques for randomizing the transition function,
- initializing, reading, writing, and rendering state values,
- scripting the iteration of CA update events,
- processing the results of computations and performing analyses.

We then go on to introduce a lattice-gas automaton, the HPP lattice gas, which is a simple fluid-dynamics model. Although LGA are technically equivalent to CA, they are not the same. This is because of the structural differences from CA. They make programing physics-like models that obey conservation laws—such as preserving global particle counts, momentum, energy, etc.—much easier.

Along the way, we see that LGA require

- defining multiple state variables per site,
- a shift operator that shifts state variables independently,
- n-input, n-output event functions.

Using the HPP LGA as an example, we see in Chapter 3.3 that some LGA and CA structures require that state variables and update events be defined on sublattices of the integer grid. In order to support these types of extensions, a programming environment requires

- a way to allocate state variables and events on sublattices of the integer grid
- a semantics for shifting state variables to alternate cosets
- a scheme for coordinatizing sublattice sites that provides a semantics for reading and writing state variables and state variable arrays.

In Chapter 4.5 we see how to program HPP on a sublattice.

Section 3.4 introduces a block partitioning CA via an example of an application that implements thermally agitated polymers. It employs a block partitioning CA to update regularly spaced blocks of sites all together in a way that allows the dynamics to locally wiggle the polymer chains without breaking or combining chains. Similar to a LGA, this BPCA and others like it can easily be programmed to microscopically, and hence macroscopically, preserve quantities. In this case the preserved quantity is the bonds of a 'polymer' chain. The polymer example requires

- an n-input, m-output local update events with events regularly spaced on a sublattice of the state-variables they update
- the ability to shift the locations of the update events,

- techniques for randomizing the update events.

While CA, LGA, and BPCA are actually all equivalent constructs and can be made to emulate one-another, as Section 3.5 points out, doing so is not altogether pleasant or simple for the programmer. Therefore, STEP architecture must allow one to program in terms of any of these paradigms.

We present examples that motivate the STEP architecture, but do not present all possible types of CA structures and variations. Instead we focus on those that are popular and practically implementable. There are an almost innumerable number of variations on the theme of CA. While they are interesting, don't have enough in common to really be part of the same architecture. In particular, they don't lend themselves to fast computer implementation.

Examples include CA on irregular grids and CA on non-Cartesian topologies including non-Abelian groups like hyperbolic spaces where the generators of CA sites don't commute. While the the computer implementation of such formats is being explored (Margenstern, 2006) they are they are not often used for CA applications. They are not too practical to implement because they have groups that grow exponentially in the number of generators rather than polynomially.

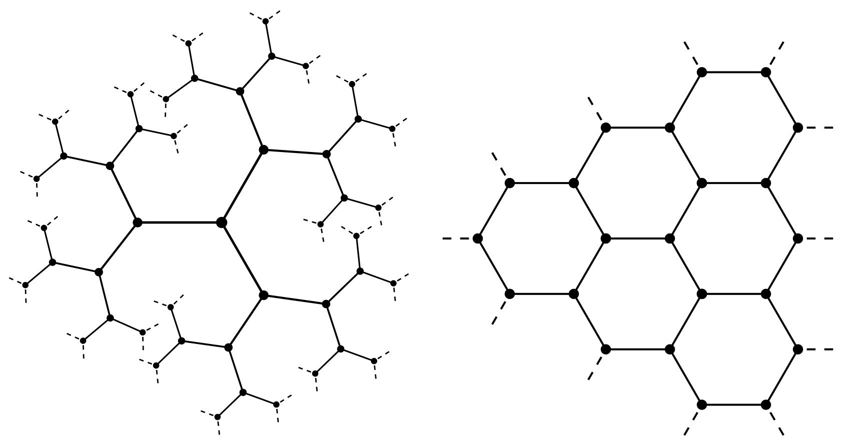


Figure 3·1: Exponential group versus a polynomial mesh

3.1 Cellular automata

This section introduces the practical aspects of programming CA by example and highlights the issues a programming environment must address to support programing and empirically studying CA. Similar to (Weimar, 1997), we present an excitable medium CA to introduce basic CA concepts. We extend the GH to a stochastic form and employ it to demonstrate a typical CA experiment.

3.1.1 A CA dynamics

Our CA example is one that implements an excitable medium An excitable medium is a spatially extended non-linear dynamical system whose dynamics supports propagating waves of excitation that, upon passing, inhibit further excitation in the same place for some amount of time. Excitable media system examples include fires rolling across a vast prairie of grass that having burned inhibit a new fire until grass grows again; contraction pulses in heart tissue; crowds doing 'the wave'; epidemiological models; and chemical reactions such as Belousov-Zhabotinskii reaction.

Such systems are locally characterized by three types of states—excited 'firing' states, refractory 'resting' states that follow and inhibit firing, and 'ready' states in which may transition to 'firing'. In 1978 Greenberg and Hastings proposed a CA excitable medium model (Greenberg and Hastings, 1978). We present a stylized version of the Greenberg–Hastings automaton (GH).

Like most of our examples, the dynamics is defined on a two-dimensional square grid. Sites on the grid are in one of three possible states—*resting*, *ready*, and *firing*. The local dynamics under GH can be stated with the following 'rule'

> *Fire* if *ready* and a neighbor is *firing*; *rest* after *firing*; and become *ready* after *resting*.

The global dynamics iterates this rule in space and time. In parallel, it repeatedly updates each state variable on the grid according to this rule. This is a general property of CA—the local transition function is uniform and applies everywhere in parallel.

The dynamics of GH is presented graphically in Fig. 3·2 where (a) represents the rule in terms of a state transition graph, (b) shows a neighborhood of local state variables that the rule might examine when deciding whether to "fire" and (c) shows four consecutive snapshots of the dynamics with this neighborhood—it evolves from a plane in the *ready* state with a single *firing* point in the middle. Macroscopically, the CA exhibits propagating *firing*-state waves that, due to the inhibitory effect the *resting* state, move forward but not backward.

Depending upon the initial conditions, different patterns emerge. A single firing point emits a propagating wave front, as in Fig. 3·2 (c).

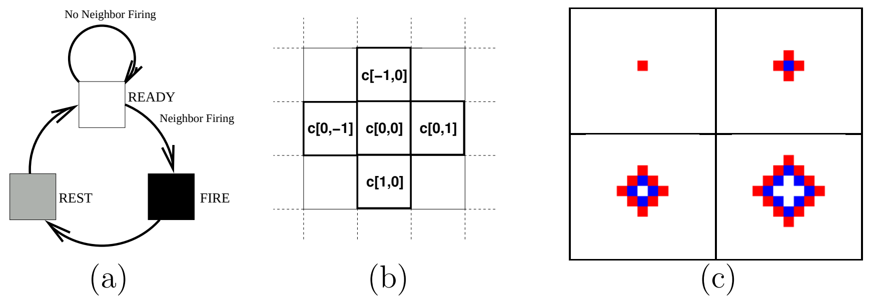


Figure 3·2: Greenberg–Hastings: A Basic CA Excitable Medium

When a programmer writes a CA program, she must actually write the local transition function in some programming language that a computer can implement. In particular, she must specify the local set of neighbor inputs and a function that maps inputs to outputs. For example, in the SIMP programming environment, which we'll be discussing, the function looks something like the following:

```
READY=0; FIRE=1; REST=2;      # Mnemonics for state interpretations

def gh():                     # Transition-function definition
  if c==READY:
   if (c[-1,0]==FIRE or c[0, 1]==FIRE or # If a neighbor is firing
       c[ 1,0]==FIRE or c[0,-1]==FIRE):
      c._ = FIRE              # transition to FIRE
  elif c==FIRE: c._ = REST    # If firing, transition to REST
  elif c==REST: c._ = READY   # If resting, transition to READY
```

A CA programing environment requires ways to

- define local CA updates
- declare the topology, its extent and the boundary conditions
- instantiate parallel state variables
- specify initial conditions (write state variables)
- perform rendering (read state variables)

3.1.2 Empirical studies on a stochastic CA

The previous example was just an introduction and starter. Let's examine a stochastic version of the Greenberg-Hastings automaton. This CA presents a more interesting topic for empirical study. We employ it to to highlight the kinds of studies a researcher may want to perform on a CA.

In the stochastic version, state transitions occur probabilistically. The new rule statement is:

> Transition to *firing* with probability p if *ready* and a neighbor is *firing*; transition to *resting* with probability q if *firing*; and transition to *ready* with probability r if *resting*.

We can interpret this dynamics as a model of a prairie fire—a *ready* site 'has grass', a *firing* site is one that's 'on fire', and a *resting* site is 'burned out'. The transition probabilities p, q, and r give the flammability, burn rate, and regrowth rate. The Poisson statistics of the transitions have an average ignition, burning, and regrowth time of $1/p$, $1/q$, and $1/r$. By modulating p, q, and r one arrives at different dynamics.

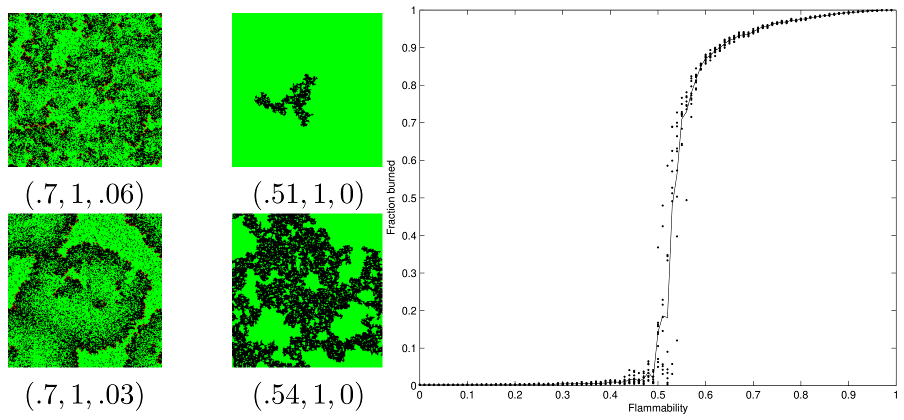


Figure 3·3: Stochastic Greenberg–Hastings Parameter Space

On the left of Fig. 3·3 we show several snapshots taken at different times after initializing with a single 'spark' in the firing state and in points in the (p, q, r) parameter space. When the flammability p is high and the regrowth rate r is low as in $(.7, 1, .05)$ and $(.7, 1, .1)$ sustained waves form and endlessly propagate over the toroidal space—in a way it is similar to the Belousov–Zhabotinsky chemical reaction.

Notice that the regrowth rate affects the wavelength. (The default boundary conditions, like those of many CA models, are periodic, wrapping around to form a torus.) When the regrowth rate is zero as in $(.51, 1, 0)$ and $(.54, 1, 0)$ a 'forest-fire' situation arises where the vegetation does not have time to grow back. The amount of forest burned depends critically upon the flammability parameter as evidenced by the amount burned in the two figures. On the right, we have used SIMP to perform a kind of percolation experiment in which we show the fraction of forest burned by a single spark over ten trials as a function of the flammability coefficient. Dots indicate the fraction burned on each trial, and the solid line marks the average over the trials.

As these examples demonstrate, in the empirical study of CA, one must define a dynamics, tweak parameters and initial conditions, compose a rendering strategy, gather statistics, run experiments over extended periods of time, and repeat them over a number of trials.

Most of the computational work is in running the dynamics. Most of the programmer effort is in defining the dynamics and how it is to be run. An experimenter cares not how the dynamics is implemented, so long as it is fast and correct.

This CA requires a few more facilities, including ways of

- probabilistic updating,
- gathering of statistics (the percentage burned),
- and repeated, scripted trials.

3.2 Lattice-gas automata

We employ a canonical, example the HPP[1] lattice-gas automaton to illustrate the general programming, structural, and implementation issues of LGA. The macroscopic dynamics of the HPP lattice gas is one that displays many of the properties of a real gas such as isotropic “sound waves”. Fig. 3·4 shows an implosion where a low-density square vacuum implodes and sends out circular, isotropic ‘shock wave’.

This isotripic behavior[2] arises from a simple, uniform dynamics based on particle collisions. Particles move inertially in discrete ‘hops’ one of four directions—up,

[1] The name HPP comes from the last initials of the model's creators, Hardy and De Pazzis and Pomeau .

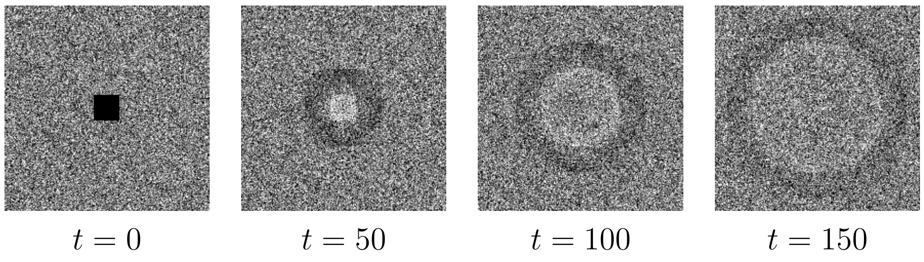


Figure 3·4: Evolution of the macroscopic dynamics of HPP

down, left and right. Each site holds up to four particles and at most one moving in each of the four directions. (Within a site, multiple particles are excluded from inhabiting the same momentum state.) Fig. 3·5 depicts the particles with arrows that indicate their momentum. In the first phase of the dynamics, the data transport/propagation phase, the particles simultaneously hop to adjacent sites as dictated by their momentum. In the second phase, the interaction phase, particles that meet head-on scatter at opposite angles in a momentum-preserving collision. All other particles continue moving according to their inertia. Fig. 3·5 shows with square boxes the sites where a head-on collision occurs and particles scatter.

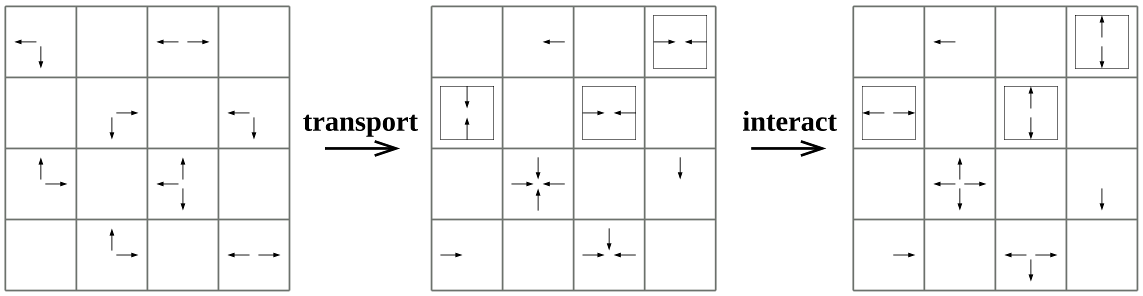


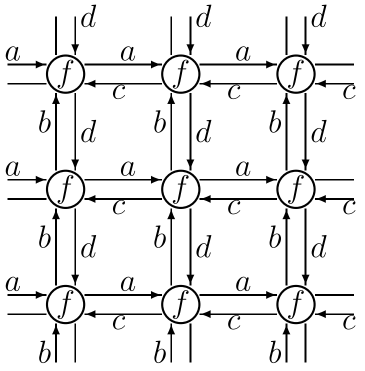
Figure 3·5: The microscopic dynamics of the HPP lattice gas

To program HPP, one requires (1) a representation of the particles, (b) a way to implement the transport phase and (3) a way to describe the interaction.

[2]Although the behavior is isotropic, a non-physical aspect of this dynamics is that it preserves momentum on each row. The FHP lattice gas is defined on a hexagonal lattice and eliminates this extra symmetry and yields the Navier–Stokes equation in the macroscopic limit(U. Frisch and Pomeau, 1986).

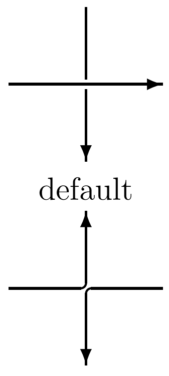
Because there is at most one particle per direction, one can represent particles with four local, binary *signals* that indicate, for each direction and each site, whether or not a particle occupies a corresponding momentum state within the site. So, we can represent the particles with four separate state-variables, well call them a, b, c, and d.

The interaction phase is relatively easy to express with respect to the state variables. It can be given as a function that maps the four signal inputs to four outputs. Because the interaction takes place within a site, there are no neighbor offsets.



in $abcd$	out $abcd$	in $abcd$	out $abcd$
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

.



(a) combinational network (b) lookup-table for f

Figure 3·6: HPP Lattice Gas

Consider Fig. 3·6 which depicts the two-dimensional structure of HPP as a combinational network. In it, HPP consists of four 'signal' arcs (labeled a, b, c, and d)—and an 'event' node where the signals interact according to the local transition function given on the right of Fig. 3·6. For most inputs, the transition function is identity, however, the ones labeled with a ∗ denote a collision where particles change tracks. The transport phase could be implemented by reading inputs from the neighbors, but it's much more logical to consider in this situation the signals to be in flight between events. As such, it's logical to shift them from one site to the next. To handle this, it would be convenient to employ a shift operator with a forward displacement v such that

$$a[\mathsf{x}]' = a[\mathsf{x} + \mathsf{v}^{<a>}]$$

where $a[\mathsf{x}]'$ is the state of signal just after the shift.

Because the LGA dynamics is uniform, shifting one signal local shifts the entire signal as a plane of values. The overall transport phase would be implemented by four separate shifts applied to each state variable. Similar to the CAM-8 we provide the shift operator as a fundamental operation in the CA programming environment. In particular, in the HPP dynamics, the shift vectors[3] would be the following: $\mathsf{v}^{<a>} = [0, 1]$; $\mathsf{v}^{<b>} = [-1, 0]$; $\mathsf{v}^{<c>} = [0, -1]$; and $\mathsf{v}^{<d>} = [1, 0]$.

From the example of the HPP lattice-gas, we have the requirement that a CA programming environment support

- a shift operator capable of uniformly translating state variables
- the capability to declare multiple, independent state variables at each site.

This means that an implementation must support having independent shifts of state-variables.

A shift operator has architectural implications as well. The CAM-8 architecture was directly built on the LGA paradigm and the user programed all dynamics in terms of transport and interaction. The transport operator applied a set of shifts applied to a binary vector of state-variables state variables. The interaction operator operated on the state-variables at each site. This arrangement has advantages.

[3]As discussed in Section 5.1, we specify coordinate vectors in $[y, x]$ format. This is similar to matrix indexes, the C-like row-major storage of arrays, but, stands in contrast to the $[x, y]$ convention of computer graphics. Similar to digits in Arabic numerals, the right-most changes the fastest.

In particular, a key architectural innovation of the CAM-8 was that a data shift need not actually be physical so long as one updates the frame of reference employed to access the data. Implementing a shift can be just a matter of incrementing an offset register that indicates the logical origin of the physical memory. The only cost of doing this is that operators that read data must add the offset to their memory addresses. Of course, the same structure can be used just as well to read CA neighbor data. The only constraint is that the transition function must prepare extra neighbor signals and shift them by the appropriate amounts to land them at the transition function sites.

CA and LGA are equivalent. Of course, if a shift operator were not available as such, one could alternately implement the dynamics via a CA that reads 'neighbor signals' data from nearby sites, and in doing so accomplishes the shift. If one didn't have the ability to just access a subset of the overall neighbor state, one could just read a more-complicated state variable that encodes them all. Nevertheless, these are annoyances that are better left to the implementation layer, rather than forcing the programmer to use just one inflexible paradigm.

3.3 Non-interacting sublattices in HPP

An interesting aspect of the way that we defined HPP is that it gives rise to two non-interacting, interlaced sublattices. Because of the way the particles move (up, down, left, and right), an individual particle will alternate between odd and even sites of the grid; particles that start out on an odd site never interact with those that start out on an even site. Take a moment to verify this by considering Fig. 3·5. In the figure, for clarity and to demonstrate that there are non-interacting sub-grids, we deliberately placed particles only on the 'odd' sites. After propagation, they all land on the even sites and would end up on the odd sites in the next iteration.

Fig. 3·7 shows the two ĉosets–the odd and even cosets—where the state variables on the sublattice land at alternating times.

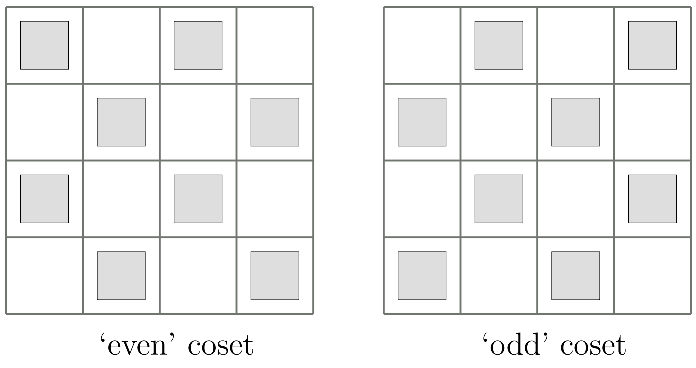


Figure 3·7: Sublattice cosets inhabited by the HPP dynamics

To see this more clearly, consider the space-time diagram in Fig. 3·8 (a). This diagram depicts the space-time network of the left and right shifts and interactions within a single row. (Actually, the structure is equivalent to that of a 1-dimensional LGA.) In (a) we have signals and interactions at each of the sites in the one-dimensional space, but, as (b) depicts, there are two non-interacting dynamics—one denoted by the network with the black events and one by the white.

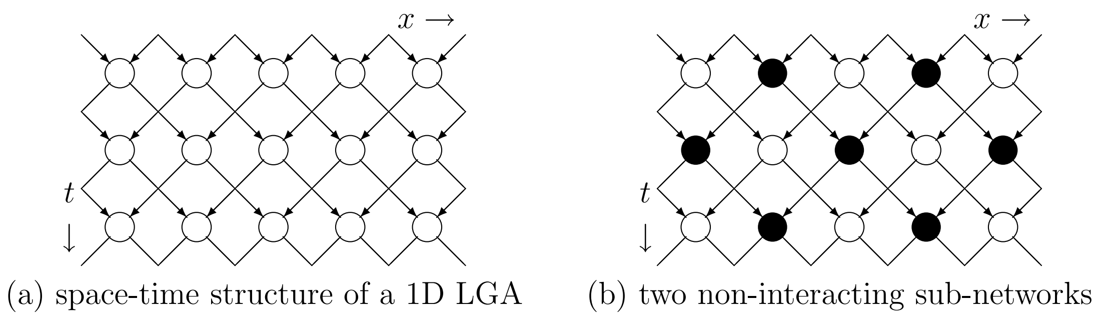
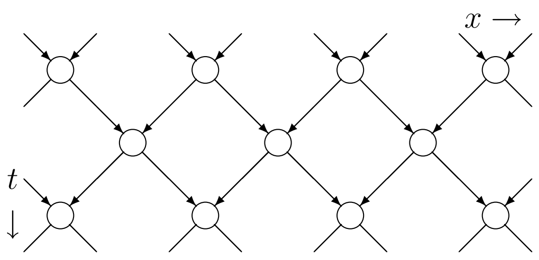
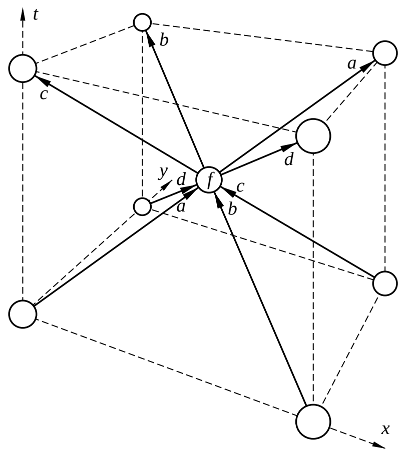

$x \rightarrow$
t
$\downarrow$
(a) space-time structure of a 1D LGA
$x \rightarrow$
t
$\downarrow$
(b) two non-interacting sub-networks

Figure 3·8: A 1D LGA with two non-interacting space-time sub-networks

Instead of computing both non-interacting dynamics it's generally desirable to save memory (memory) and time (computation) by implementing just one. Fig. 3·9 (a) depicts a space-time structure of events and signals in the 1D LGA with only one of the sub-networks. Fig. 3·9 (b) depicts the unit cell of the sub-network in the 2D HPP case. The body-centered cubic (BCC) unit cell depicts the odd-coset/even-coset alternations in time. Indeed, the sites inhabited by signals are those where $(x + y + t)$ is odd.



One-dimensional LGA sublattice



BCC space-time unit cell (HPP)

Figure 3·9: Space-time 'crystal' of two lattice gases

Now, the question is how can one represent and program such a structure. In CA programming practice, this is not directly possible. Most programming environments do not have a way to allocate variables on sub-lattices. One must instead program at a lower-level by packing state variables into a dense, orthonormal grid that corresponds more directly with the storage layout of the data in computer memory. This often requires specifying the dynamics with multiple phases and skewed local or global coordinates. It entails much more programmer effort.

In our work, we strive to do better. In principle, the space-time network of HPP can be described in terms of state-variables that are allocated on a *sublattice* of the grid and are shifted by the LGA shift operator to alternate cosets. The CA events would also need to be allocated on the same sublattice and shifted before being applied. Later in Chapter 4.5 we introduce constructs for programming with sublattices.

Although the exact implementation is something that the programmer would rather have the computer do, the computer still has to do it. In Chapter 7 we demonstrate methods converting sublattice descriptions into software.

A programming environment that handles sub-lattice LGA in a generic way (supporting various types of sublattices and not just the one shown here) requires a suite of conceptual, programming and implementation strategies. From a conceptual and programmatic standpoint the programmer needs

- facilities for describing arbitrary integer sublattices of the square grid,
- a semantics for shifting sublattices to alternate cosets,
- ways of representing the cosets that are more general than the odd/even description given here,
- indexing conventions for sub-lattice allocations,
- rendering techniques that respect the logical geometry of the problem,
- techniques for verifying that this is compatible with the global topology (e.g. that the lattice wraps on the torus rather than skipping to another coset)

From an implementation standpoint, one needs

- a memory allocation strategy for sparse signal allocations on sublattices,
- ways of converting logical coordinates into indexes into the storage allocation,
- update scans that respect the current coset that signal inhabits.

3.4 BPCA example: 'wiggling' polymers

Let's now introduce the concept of a block partitioning CA (BPCA) with a 2-dimensional model of thermally agitated, wiggling polymers. Fig. 3·10 shows the macroscopic dynamics. Initial, unlikely polymers in Fig. 3·10 (a) relax into more likely ones under thermal-agitation as shown in (b), and, relax even more after an even longer time in (c).

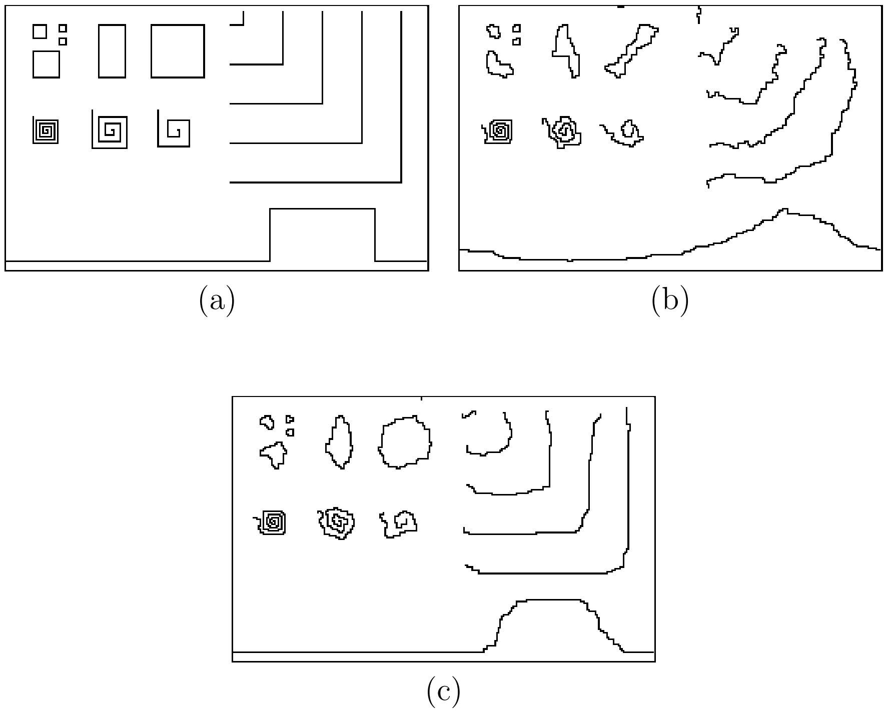

(a) (b) (c)

Figure 3·10: Evolution of thermally-agitated polymers

In this model, polymer particles are represented by binary state variables on the grid. If two particles are adjacent horizontally or vertically (but not diagonally), they are a bound polymer string. Fig. 3·11 (a) depicts a configuration with six polymers. The microscopic dynamics acts locally, wiggling end-chain particles and kinks. Specifically, it moves particles diagonally so long as doing so does not break or create a polymer bond. Fig. 3·11 (b) marks with arrows some proposed local moves that, when applied in isolation, don't create or break a polymer bond.

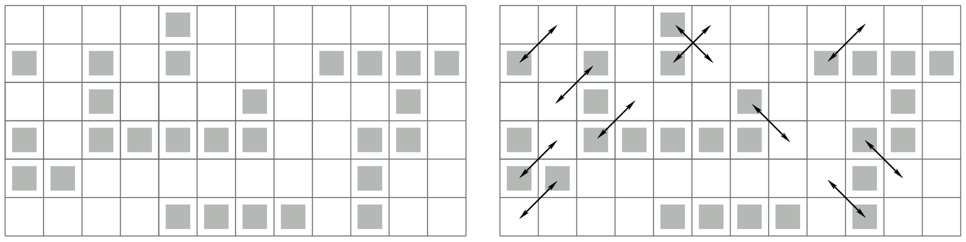


Figure 3·11: Fine-grained polymer dynamics

Fig. 3·12 demonstrates a simple mechanism for deciding whether a diagonal swap is allowed or denied. A diagonal swap is allowed if and only if the sites highlighted with dotted lines in Fig. 3·12 are empty.

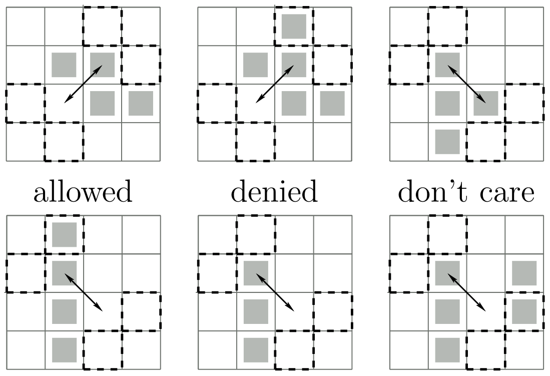


Figure 3·12: Examples of proposed moves

Now, in order to cast this as a parallel CA-like dynamics where updates are performed in lock-step parallel, we take the basic stencils of Fig. 3·12 and combine them into a single operation that updates a block of four sites together. The form of the stencil is that of a "swiss cross" The inner part is the part that is subject to change while the outer is the part that the change depends on. This part should not be changed while the part in the center is being updated.

When updating the dynamics, the polymer BPCA updates non-overlapping blocks

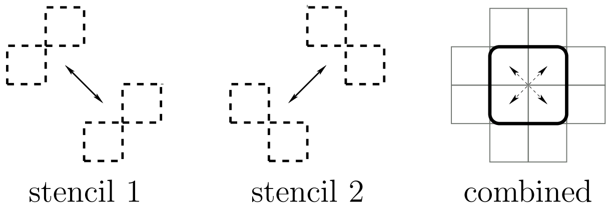


Figure 3·13: Combined stencil for polymer updates

and ensures that the areas upon which other updates depend do not change. Fig. 3·14 shows a partitioning of the space that guarantees this constraint.

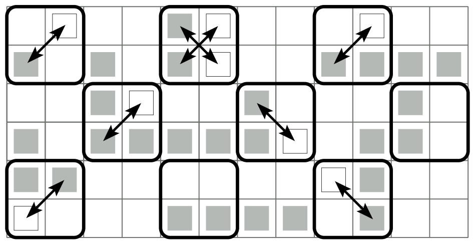


Figure 3·14: Polymer update event stencil and partitioning according to the stencil

The dynamics updates by shifting the blocks being updated to the eight different possible coset positions—this gives every particle a chance to wiggle.

In order to support a dynamics like this, we require

- events defined on a sublattice,
- a way to update multiple grid sites with one event,
- a mechanism for shifting the update events to alternate cosets.

3.4.1 Lattice-Boltzmann LGA and continuous-valued CA

The *lattice-Boltzmann* approach is a 'bulk' version of the LG approach in which *particle-level* updates are replaced updates involving particle counts. The structure of the computation remains similar; however, the complexity of the state and the update function has increased, while the number of nodes has typically decreased.

An advantage of lattice-Boltzmann techniques is that substantial numerical accuracy can be preserved. Structurally, lattice-Boltzmann techniques are equivalent to the LGA we've already seen. Local shifts are often employed to move data. The only difference is that integer types are required.

A related format, *continuous-valued CA*, employs CA that have continuous state-variables implemented by floating point numbers. This type of CA bears similarity to finite-difference methods for solving partial differential equations and methods for performing multi-dimensional convolution. However, in addition to performing local linear operations, the iterated dynamics can perform arbitrary non-linear transformations. For example, one can implement reaction-diffusion systems by combining reaction and diffusion operators into the same CA transition function. The diffusion operator can be implemented with a sum of the products of local values and weights given by a truncated Gaussian convolution kernel. The diffusion operator can be implemented via an arbitrary, non-linear function.

To support these formats, we need only add support for integer and floating point state-variable types. This is often not supported in CA programming environments because doing so requires compiling the transition function to machine code rather than just implementing it as a table lookup.

3.5 Equivalence of CA and LGA and BPCA

We've introduced CA, LGA, and BPCA by example. We've seen that

CA have a dynamics defined by an n-input, single output *event* function that reads neighboring states values and outputs the next state at each site;

LGA have multiple state variables and a dynamics defined by (1) an n-input, n-output *event* function that reads and writes the set of state variables at each site and (2) a *transport* (shift) operator that moves data among the sites[4];

BPCA have one state variable and a dynamics that employs events that update blocks of state-variables together through an n-input, m-output local transition function that reads and writes local state variables within a block.

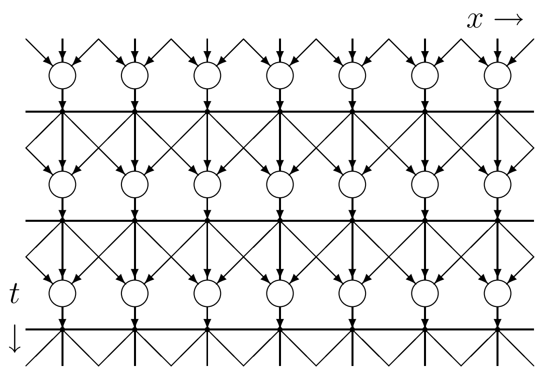
Despite the surface differences, we call all of these forms of computation 'cellular automata' because they are all have a spatially distributed, parallel, uniform computational structure. As such, it turns out that these forms are all equivalent and can be mapped into one-another.

Regardless, of the fact that they can be mapped into one-another, we choose to support all three in the STEP architecture. We do this because (1) this makes programming simpler—mapping one form into another is unpleasant work and makes programs more confusing—and (2) the different expressions provide useful implementation hints.

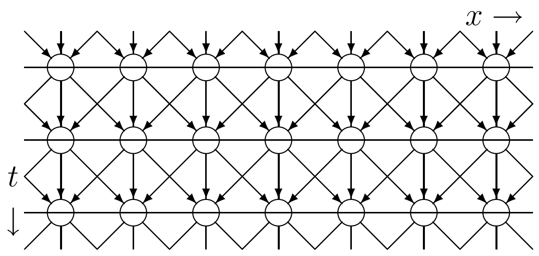
Let's first consider the equivalence of basic CA and LGA. It's always possible to map one form into the other. Fig. 3·15 depicts two equivalent CA/LGA structures. Notice that there are two primary differences. The LGA has an n-input, n output transition function while the CA has a n-input, single output transition function and a 'fanout' where a state variable is read multiple times.

To map an a CA into a LGA one just needs to (1) allocate extra state variables for each neighbor that the transition function sees, (2) modify the local event so that it performs a fanout by writing the same data to each output and (3) perform a shift the outputs to neighboring sites. In fact, this is how one implemented CA in the

[4]These operators are sometimes also called the collision and propagation operators. As has been noted (Hénon, 1989), its useful to preserve the distinction between LGA and CA because in representing a LGA as a CA one must break the symmetry of the LGA operations by combining the transport and interaction operations, either performing transport followed by interaction or interaction followed .



CA space-time network



A LGA version of the same network

Figure 3·15: A CA and an equivalent LGA

CAM-8, which was based purely on LGA like operators.

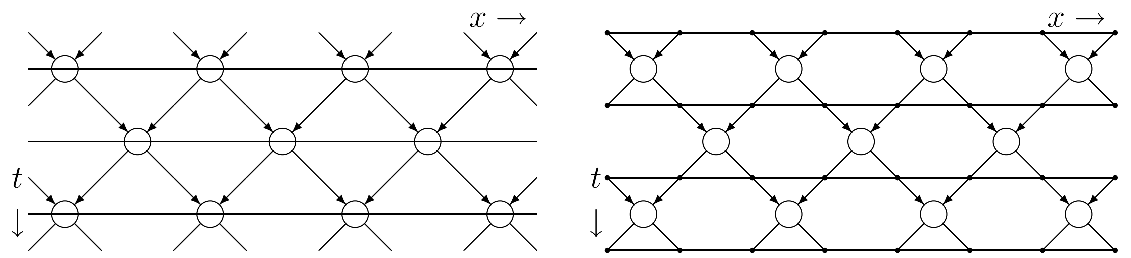
To implement a LGA in a CA, one must expand the transition function so that it inspects the states of the neighbors as specified by the LGA shift and expand the state set to include a state variable for each LGA state.

From an implementation standpoint, the CA version suggests a 'double-buffering' strategy where the input state is present while the output state is being written. Double buffering allows the input state to be read multiple times.

We prefer to keep both descriptions around because, if CA as a LGA because doing so requires employing extra state variables to store what is logically just a fanout of a CA state variable. It's better that the implementation know that a fanout is what is needed and decide whether to implement it by allocating extra storage (this is what the CAM-8 would do) or to just read the same state variable multiple times. One advantage of the LGA approach is that the local event can process data in-place.

Fig. 3·16 depicts an equivalent LGA and BPCA space-time network. The difference between LGA and BPCA is that a BPCA applies on a sublattice of the state variables, partitioning state variables into blocks that are updated together and assigning different roles to each *coset* within a block. A LGA on the other hand, associates each state variable with an independently moving signal. In a LGA, the state is visible just before and just after an event. In a BPCA, the state is visible between events.

From an implementation standpoint, the difference between the LGA and the BPCA is that in one form, one has a set of state variables and in the other, the state variables are all in one logical storage structure and simply partitioned by the events. It is useful to be able to describe a BPCA state variables as a single array rather than a state variable for each coset because (1) doing this is awkward, and (2) it does not preserve information about the spatial relationship among the state variables.



(a) LGA space-time network (b) A BPCA version of the same network

Figure 3·16: A LGA and an equivalent BPCA

3.6 Implementing CA

Virtually all computer architectures can be made to implement cellular automata. Each architecture engenders its own set of implementation techniques. Parallel architectures are particularly effective because the "embarrassingly parallel" nature of the CA paradigm maps well onto parallel architectures.

Even within one architecture, there are a variety of implementation strategies. For example, on a single-processor computer one can employ sparse updating techniques, use vector-processing operations, and make optimizations like implementing transition functions as lookup tables.

Architectures and implementation strategies also have somewhat variable performance constraints and requirements. For example, FPGA and related implementations may be able to process a sequence of updates by pipelining them. Others may want to combine certain operations (the CAM-8 combined shifts and local data interaction). In order to perform pipelining and any loop fusion that may be involved with such a technique, an implementation would like to know, in advance, a batch sequence of updates to be performed. We support this in the STEP architecture with a compiled *Sequence* operation.

When it comes to implementing a cellular automaton, there are a number of ways to do it. It is essentially data-parallel computation. As such, it is possible to do it on a regular PC by making a loop that scans the space and does updates. In parallel in a shared memory context where multiple processors scan subsets of the space.

If one would like to use multiple machines, one can partition the space, do an update and compute edge data as needed.

CA are also quite amenable to vector processing. In vector processing, a processor loads vectors of words at the same time.

When the local function f has a small number of possible input state values such as 20 bits (less than 2 million possible states), one may use a lookup table to do processing.

Attractive computer implementations include serial, multi-processor, multi-computer implementations. Attractive hardware implementations include vector processing architectures like the CAM-8 and reconfigurable hardware approaches of FPGAs. Implementation strategies range from "crystalline" implementations where one maps each local computation to its own spatially distributed processor, to serial scans performed by a single-CPU computer. Scan techniques include data partitioning, multi-update pipelining, vector processing, block updating.

In this type of data-parallel loop one can perform loop fusion and fission, in both the spatial and temporal dimensions.

Implementations must decide whether to store state variables separately or all-together. Whether to double-buffer the state or perform operations in-place with a buffer.

We highlight various implementation strategies of relevance to these architectures.

Shifts can be implemented without actually moving data. One can shift the frame of reference for the state variable, rather than the data itself. To implement this, the straight-forward thing to do is to employ one of the central innovations of the CAM-8, namely, create an offset register that indicates the logical, shifted location of the origin in physical memory. So long as the offset is added to every memory reference, it will seem that the data actually has been shifted.

3.6.1 Fine-grained, massively parallel

Conceptually, the most straight-forward implementation of a given CA is the one that the paradigm itself suggests—an indefinitely extended array of homogeneous, locally connected processors. One could imagine implementing this in some physical substrate like a crystal or mapping it onto silicon.

At the moment, these types of approaches are not practically feasible, but that may change with advances in self-assembly and technologies like quantum-dot CA.

To see why, consider what would happen if one tried to implement a CA directly in silicon. Although one could create a direct implementation in silicon where each local CA update and state variable is implemented with its own locally-interconnected processor, there are several practical problems with such an approach using current technology. First, the topology would be limited to that implementable in current 2D chip fabrication architectures. CA with topologies with large sizes in the third and higher dimensions can not be implemented in this way. More practically, though, such a locally connected architecture would likely suffer from a lack of generality—not all local interconnection topologies might be possible. Also, a direct implementation in silicon would consume too much power.

With just one one current CMOS die, one could implement a nearest-neighbor 2D CA with billions of sites and capable of updating billions of times per second. The calculation rate would on the order of 10^{18} sites-per-second. Too bad that there's no free lunch. Doing this would also lead to power consumption that is easily in the thousands of Watts. Because the chip would be made up of many small processors, each switching, charging and discharging the gate capacitors that store the state and implement logic functions, the power consumption would far exceed most chips which include lots of inactive gates such as RAM.

Nevertheless, if one did arrive at a suitable fine-grained implementation, performing random memory access would likely be rather expensive because, unless another network were put in place to handle it, all data would need to be shifted in and out locally. The local interconnection topology may be limited and implementing large shifts might be complicated. The boundary conditions might not be adjustable and it's unlikely that they would wrap-around—most likely, there would be fixed or 'reflecting' boundary values. The size would likely be fixed, and there might be restrictions on state variables.

More likely than not, a physical substrate would run just one CA program. The role of the STEP architecture is more of simulating such a machine before it is implemented.

3.6.2 Serial

The most commonplace implementations of CA are those on single-CPU computers. When implementing a CA on a CPU, updates are performed by scanning the state variables.

Most CA scans are serialized at some level and individual site updates are implemented by time-sharing a single processor that scans sites to perform the update. In effect, the logical processing of the CA model is implemented by time-sharing a complex CPU.

In this kind of implementation, a program loops over all the sites applying the local interaction to each. Shifts can be applied immediately buffered stored as an offset to each state variable that is added to subsequent operations.

When a rule is simple, i.e., it involves less than 16 or so bits of state variables, a simple way to implement is via a lookup table. The cost incurred is that of any necessary shifting and masking to pack bits into a table lookup plus the cost of performing the lookup, plus shifting and masking to write the LUT outputs.

Handling boundary conditions can be expensive unless care is taken to avoid expensive modulus operations.

3.6.3 Data parallel

The implementation strategy here is to either partition the space into tiles divided among several processors which perform scans within a tile and communicate at the edges of the tiles.

This works well in multi-CPU and multi-computer environments. Even with a single processor, breaking up the scan into several scans that operate on blocks of memory can help improve data locality in the second and higher dimensions.

Disadvantages of this approach are that one has to partition data and arrange for communication at the edges. In higher dimensional spaces, most of a tile would be at the edges sync (in higher dimensions, "an orange is all peel") amount of communication increases with number and distance of 'neighbors'. It may be complicated to read and write data regions, especially when they straddle tiles. Latency may be unexpectedly high when a tile is on a remote computer.

In data-parallel, SIMD architectures like STEP, parallelization is most often carried out by partitioning data into blocks to be updated in-parallel by separate computing modules (processors)(Culler and Singh, 1999). Each module is responsible for computing updates within its own block and communicating with neighboring blocks with shared edges. The crystalline structures we implement here are ideally suited for such a data-parallel implementation strategy(Worsch, 1999). In the STEP architecture, it is the job of the back-end to decide how to partition data among available computing resources and to maintain the logical memory model at the STEP interface.

3.6.4 Vector processing SIMD

In a vector-processing CA implementation, rather than updating one site at a time in a scan, the scan loads inputs, computes outputs and writes outputs for a whole vector of sites simultaneously. Typically,

Performing updates via vector processing requires that the transition function applied at each site be expressed as a uniform sequence of vector operations. This requires the "flattening" of any conditionals in the transition function. Some vector processor languages handle this with "vector masks" that condition whether each vector instruction is performed. In the CAM-8 the flattening was accomplished by converting the transition function into a lookup table.

Combinational networks of bit-wise defined logic functions can be implemented very efficiently using a method called *multi-spin coding* (Jacobs and Rebbi, 1981). Multi-spin coding was first used in Monte-Carlo simulations of spin systems and programmed by hand in an ad-hoc fashion. The basic idea is to separately pack the bits of the state-variables into data words; each state-variable bit of a site is stored separately and packed across sites into words that serve as multi-site vectors. Then, the update function specified by the bit-wise combinational network can be computed simultaneously for as many vectorized sites as there are bits in a word by using ordinary ALU instructions.

In his work on the JCASim environment, Weimar nicely explains some of the different CA implementations that are possible (Weimar, 2003). In particular, he notes that using the "multi-spin" coding technique it is possible to make a cellular automaton run faster in Java than in the C programming language. Weimar has automated (Weimar, 2003) multi-spin encodings of CA having simple combinational networks and thereby achieved update speeds two orders of magnitude faster than the LUT approach.

Suitable hardware implementations come in a few basic forms—mesh computers, special-purpose vector-processing architectures like the CAM-8 and FPGAs. Standard computers also make for nice CA implementation platforms.

- requires memory system and storage allocation suitable for loading and storing vectors
- because a vector may span a boundary, special precautions must be taken there
- in simplest implementation, input and output arrays required PC can use MMX, or even binary vectors (multi-spin)
- depending on the available vector operators, there may be restrictions on the form of the transition function.

Stream processing and pipelining

The fact that CA are iterated in time as well as space make them an ideal candidate for stream processing. In this strategy, the processor streams state of a local region into a pipeline that performs a whole sequence of updates to a local region, one after another subject to data-dependence constraints. Of course, a stream processor must respect data dependencies, so there is a limit to how many times the processor can update a region before moving on to the next.

The approach relies on the fact that entire sequences of updates is performed all at once. As such, for efficiency, it requires that the applications layer "batch" as many updates as possible.

Properties

- knowing the computation to be done ahead-of-time is necessary for constructing the patch
- optimizing the pipeline may be expensive and switching from one update sequence to another may be expensive.

3.6.5 Sparse updates

When only a small subset of a CA space has active computation in the sense that state ensembles are evolving in a non-trivial fashion, it may be reasonable to employ adaptive strategies that only update the active region and do some some simplified processing on the inactive regions.

An example of this kind of approach are Gosper's "hash" implementations of Conway's Game of Life (Gosper, 1984). This CA is highly dissipative and quickly leads to configurations that oscillate with relatively short orbits. In such situations, it may be possible to "hash" a local local configurations for patches and employ an update that looks ahead arbitrarily into the future for the patch.

The patch would need to be updated in a non-uniform fashion as soon as activity impinges on it from a neighboring patch.

This type of implementation lends itself well to emulating infinite topologies that have a uniform pattern. If the pattern is uniform, then each "tile" of a certain size is exactly the same as all of the others. Only one uniform "tile" need be stored or updated. If the CA contains "active" regions that deviate from the uniform state, then only that volume, plus that of one repeating tile need be actually stored or updated. Of course, most "chaotic" or "complex" CA dynamics are totally unsuited to this style of computation because the size of any initial active region would grow "without bound". However, one could, of course, program boundaries into the dynamics to contain the active regions.

This type of implementation requires a general-purpose computer that can examine a tile. Nevertheless it is compatible with parallelization strategies. It may also require capabilities for specifying "infinite" boundaries and the ability to initialize state variables at allocation time to infinite, repeating patterns.

3.7 Implications for the STEP architecture

At the user programming level, we've seen that we would like to support CA programming environment that includes CA events, LGA shifts, and updates on sublattices. The programs should gracefully handle rendering and viewing of results and provide means to perform random access on state variables. The architecture must have a notion of CA 'events', state variables, sublattices, and support various types ranging from small binary types through floating point numbers.

The transition function of a CA event must be able to operate on blocks of the same state variable as well as multiple state variables.

The CA events and state variables must be able to be declared on sublattices.

It must be possible to implement various boundary types—periodic boundaries are the most natural, though.

Rendering state variables must be supported directly.

CA implementation techniques range from special-purpose hardware to various software techniques. Across all of them we have the following general properties that an architecture for computing with CA must bear in mind.

- the state may not be local,
- accessing state variables may involve a high latency,
- it's possible to pack data in a variety of ways for performance, so the actual storage should be maintained by the machine,
- it's better to know ahead-of-time what operations may be performed,
- operations may be combined or re-ordered,
- some implementations may have specific constraints and may not be able to implement all structures and constructs,

We would like to program at a high level and have the machine efficiently implement the programs. Programming in batch-mode is not acceptable—it must be possible to interactively control and script CA computations. While it will be possible to specify some sequences of updates ahead-of-time, it will not be possible to batch everything.

We want a high-level interface to CA and we also want the programmer, in the same environment, to have high-level facilities for tasks related to the CA itself. This includes preparing initial state variable configurations, performing analysis on output configurations, saving data, constructing user-interfaces, and the like.

We chose to implement the architecture in the Python programming language because it is a well-supported, popular, well engineered language with a simple, expressive syntax, a vast set of user libraries, and extension capabilities that allow one to call external libraries written in C. This latter part is important because we need STEP implementations to run efficient machine code. Of particular use is the numarray[5] extension module, which provides a useful multi-dimensional array class that we use extensively for representing state variables at the user level.

[5]See `http://www.stsci.edu/resources/software_hardware/numarray` and `http://numeric.scipy.org/` for more on the Python array modules.

Chapter 4

The SIMP Programming Environment

On the surface, SIMP is an extension module for the Python programming language that supports writing CA applications. We have attempted to embed a significant amount of knowledge about programming CA into SIMP. As such, SIMP is a good place to start in explaining our overall architecture for computing with CA.

The purpose of this chapter is to demonstrate the STEP architecture, showing how it may be employed to tackle the CA applications described in the last section. Along the way, we introduce the STEP processing model, elements of the STEP API, and the SIMP conveniences that make programming atop the API relatively easy. SIMP is efficient because it's based upon the STEP API. Later in Chapter 5 we detail the STEP API and how SIMP extends it.

The sections of this chapter cover the topics of programming CA, LGA, and BPCA in the same order as these topics were presented in the last chapter. Recall that, CA update sites as a function of their neighbors; LGA employ two operators, a transport operator (shift) and an interaction operator (event), to perform an update; and BPCA update blocks of sites all-at-once. In Chapter 3 we employed examples—the Greenberg-Hastings CA, the HPP LGA, and the polymer BPCA—to introduce the three formats. In this chapter, we'll see how one may program them in SIMP. Additionally, we'll see how one may program and render the space-time history of a 1D CA and re-program HPP as a BPCA using the Margolus neighborhood.

The first section, presents the full program for the Greenberg-Hastings CA, including techniques for defining, running, scripting, and visualizing this dynamics. The introduces randomized CA and general facility that STEP provides for writing efficient randomized dynamics—the stir operation. Following that is a description of how SIMP handles one-dimensional CA and allows one to render space-time histories. The next section introduces the `Shift` operator for programming lattice gas automata.

Conceptually, programming lattice-gas automata and block partitioning cellular automata often requires lattices other than the one represented orthonormal square grid of the integers $\mathbb{Z}^n$. STEP and, by extension, SIMP provides novel constructs for programming non-orthonormal lattices. We interoduce these abstractions in

Section 4.5, which demonstrates how to program LGA and BPCA applications using sublattices, is probably the most interesting section in this chapter. In explaining how to program HPP on a sublattice it demonstrates how SIMP programs allocate, update, render, and read and write state variables on sublattices. It also shows how one can employ the popular Margolus neighborhood BPCA technique to program HPP as a BPCA. The last section shows how to program the Polymer BPCA.

4.1 A SIMP example—Greenberg-Hastings CA

We start by explaining how to program the Greenberg Hastings CA discussed in Section 3.1.1 and presented in Fig. 3·2. Most SIMP programs follow the same pattern as the file, ‘greenberg_hastings.py’, which we list in entirety below. Take a moment to glance over it. We'll address the details in the sections that follow.

```
from simp import *             # Import simp and helpers

# ----------------------------------------------- GEOMETRY AND STATE
Y,X = 200,200
initialize(size=[Y,X])         # Declare an YxX square grid
c = Signal(SmallUInt(3))       # State variable (signal) declaration
READY=0; FIRE=1; REST=2;       # Mnemonics for state interpretations


# ----------------------------------------------- DYNAMICS
def gh():                      # Transition-function definition
  if c==READY:
   if (c[-1,0]==FIRE or c[0, 1]==FIRE or # If a neighbor is firing
       c[ 1,0]==FIRE or c[0,-1]==FIRE):  #
      c._ = FIRE               # transition to FIRE
  elif c==FIRE: c._ = REST     # If firing, transition to REST
  elif c==REST: c._ = READY    # If resting, transition to READY

gh_event = Event(gh)             # Create transition-function Event


# ----------------------------------------------- RENDERING
red,green,blue = map(IoSignal,[UInt8]*3)  # Declare color outputs

def tricolor():                # Function describing a color map
  if c==READY:
      red._=green._=blue._ = 255 #   READY => white
  elif c==FIRE:
      red._   = 255              #   FIRE  => red
  elif c==REST:
      blue._  = 255              #   REST  => blue

# Create a rendering Event and renderer (red, green, blue are outputs)
tricolor_rend = Renderer(Event(tricolor),(red,green,blue))
```

```
# ---------------------------------------------- INITIALIZE
c[:,:] = READY                # Initialize all sites to READY
c[Y/2,X/2] = FIRE             # Set site in the center to FIRE


# ---------------------------------------------- USER INTERFACE
ui = Console(tricolor_rend)   # Instantiate a console called "ui"
                              #   and initialize its renderer
ui.bind("DYNAMICS",gh_event)  # Bind console's DYNAMICS handler
ui.start()                    # Start the interactive interface
```

The script is divided into five sections: *geometry and state*, *dynamics*, *rendering*, *initialization* and *user interface*. The very first line in the code imports the SIMP programming environment from the `simp` module. The geometry and state section first sets up the topology in which the CA will be defined and declares the CA's state variables. The dynamics section declares the event that implements the dynamics. The rendering section describes how the state is visualized. The initialization section sets up the initial state. The user interface section declares a `Console` object for interactively running the dynamics and viewing rendering results on-screen.

Let's first address the `import` statement in the very first line. It is a Python statement that imports `simp` and all of its definitions.[1] `simp` contains various constructs—methods, functions, and constructors—that the rest of the program uses. Additionally, it maintains global variables and defaults used by these constructs.

4.1.1 Geometry and state

Calling `initialize` initializes SIMP's global variables. In particular, `size` parameter declares the size of the two dimensional grid where state variables will be allo-

[1]Although SIMP is designed to be used this way rather than as a module imported with '`import simp`', the latter will work, but all `simp` definitions must be qualified as in `simp.initialize`.

cated. In the example, the size is 200×200 and is stored two variables–`Y` and `X`–for later use. In general, the value of each element declares the size of the grid in that dimension while the number of elements in the size vector specifies the number of dimensions. While `size` is the only parameter that SIMP needs for an ordinary CA, `initialize` also has optional parameters. Among other things, they specify the default lattice generator matrix and the runtime space-time event processor (STEP) implementation to be used.

State declaration

SIMP state variables are called *signals* and are `Signal` object instances. The line '`c = Signal(SmallUInt(3))`' allocates a signal with a ternary integer state set $\{0, 1, 2\}$. `c` has a ternary value at at every point on grid—it's basically a 200×200 array. For convenience, the code also creates some mnemonic names—`READY`, `FIRE`, and `REST`—for each of `c`'s possible values.

4.1.2 Dynamics

The dynamics of a CA is defined locally by a transition-function that is evaluated everywhere in parallel. In the dynamics section, the code

```
def gh():                    # Transition-function definition
  if c==READY:
   if (c[-1,0]==FIRE or c[0, 1]==FIRE or # If north, east, south,
       c[ 1,0]==FIRE or c[0,-1]==FIRE):  #  or west is firing.
      c._ = FIRE             # Transition to FIRE
  elif c==FIRE: c._ = REST    # If firing transition to REST
  elif c==REST: c._ = READY   # If resting transition to READY
```

defines the transition-function for the Greenberg–Hastings event. The statement '`gh_step = Event(gh)`' uses the transition-function `gh`—which is just an ordinary

Python function—to create a parallel CA `Event` object that, when called (as in `gh_step()`), applies the transition-function to all sites on the grid in parallel.

An `Event` object packages a transition-function into a STEP operation—an object that a back-end STEP implementation can execute. The object itself describes the operation—the transition-function, the context in which it was created etc., while calling it tells the back-end to execute the operation. A SIMP program may use a `Event` object directly by calling it to execute the transition function or it may pass it as an argument to other constructs, such as the `Console` and the `Renderer`, that may need it. A `Event` is not the only type of STEP operation—`Shift`, `Sequence` and `Stir` are other examples that appear later in this chapter and in Chapter 5.

Now, a transition-function locally maps current-state input values to next-state output values. Inside a transition-function, signals are accessed locally, therefore, rather than referring to the entire parallel data allocation, as is normally the case, accessing a signal name inside a transition-function references its value *at the site being updated*. For example, to check whether `c` at the site being updated is currently in the `READY` state, the transition-function uses `c==READY`. To write the output value of a signal, a transition-function assigns values to a signal's output attribute—the underscore attribute. For example, to set the the next state value of `c` at the site being updated to *firing* the transition-function makes the assignment '`c._ = FIRE`'.

`gh` looks at the *von Neumann neighborhood* of `c`—the site itself and its neighbors at an offset of ± 1 in the Y and X directions as shown in Fig. 3·2 (b). The neighbor value of `c` to the right is `c[0,1]`, to the left is `c[0,-1]`, above is `c[-1,0]`, and below is `c[1,0]`. SIMP subscripts are listed from the most significant to the least[2]; therefore the subscript in the higher dimension, Y, comes before that of the lower dimension, X. In accordance with the conventions of computer graphics—Y grows downwards while X grows rightwards (Z grows away from the viewer). (Note that, within a transition-function, neighbor values are referenced by relative subscripts. Outside of transition-functions, subscripts are absolute coordinates.)

`gh` does not always assign an output value of `c`. For example, if the current state is *ready* and no neighbors are firing, the event does not write a next state value. What, then is the next state value of `c`? Of many conceivable defaults, our choice in SIMP was that the default new value of `c` is the previous value of itself. Thus, the event specifies that if no neighbors are *firing* and `c` is *ready* then `c` remains *ready.*

4.1.3 Rendering

The rendering section declares how to convert state information into images that can be displayed. Like the dynamics, rendering behavior is defined by an `Event`. Unlike the dynamics, however, the `Event` does not output to ordinary signals. Instead, it writes to special `IoSignal` objects—they are just like `Signal` objects, except that they are write-only and need not store state. SIMP also supplies special `Renderer` classes to manage the conversion and output of rendering data to image arrays. The `Renderer` provides an interface that's suitable for on-screen display by a `Console` user interface.

The output of a rendering event is an array containing color information. Color channels are encoded with 8-bit color `IoSignal` objects of type `UInt8` (8-bit unsigned integer). The `UInt8` type is necessary because the `Console` expects images encoded as 8-bit values in which 0 is the minimum intensity and 255 is the maximum intensity. Conceptually, the simplest way to declare the necessary `IoSignal` objects for color rendering is as follows

[2] As of version 0.6 this convention replaces the prior least-significant-first convention inherited from physics and linear algebra. Although the change was precipitated by the adoption of the `numarray` package for handling multidimensional arrays in Python, the most-significant-first convention is more natural when subscripts are interpreted as a generalization of positional notation for numbers—subscripts, like digits, are usually ordered from the most significant to the least.

```
red = IoSignal(UInt8)
green = IoSignal(UInt8)
blue = IoSignal(UInt8)
```

However, we use the following more succinct form

```
red,green,blue = map(Signal,[UInt8]*3)
```

The code `[UInt8]*3` is Python shorthand for `[UInt8,UInt8,UInt8]`, a list with three references to the `UInt8` type. The `map` is a built-in Python function that, in this example, calls the `IoSignal` constructor on each of the three list elements and returns a list with three new `IoSignal` objects. Finally, Python list comprehension handles the assignment of `red,green,blue` to the three signals returned by `map`. In this example, and many others, mastering Python helps one to master SIMP.

The rendering function was declared as

```
def tricolor():               # Function giving a color mapping
  if    c==READY:
      red._=green._=blue._ = 255 #   READY => white
  if    c==FIRE:
      red._   = 255              #   FIRE  => red
  elif  c==REST:
      blue._  = 255              #   REST  => blue
```

Similar to the event's transition-function, the rendering function takes advantage of default values. This time, it takes advantage of the fact that the default value of an `IoSignal` is 0. Therefore, when '`c==FIRE`', `red` is set to `255` while `green` and `blue` have a value of `0`.

After declaring the color-map function the program creates a `Renderer` object as follows

```
tricolor_rend = Renderer(Event(tricolor),(red,green,blue))
```

The first parameter is the rendering event, the second is the set of outputs that are rendered. The tuple '`(red,green,blue)`' gives the ordering of the output signals in the array that results from the rendering operation. We give them in this order because the `Console` expects RGB arrays.

4.1.4 Initialization

This section uses signal subscripts to assign initial values to the signals. As previously mentioned, the subscripts here are global.

```
c[:,:] = READY                  # Initialize all sites to READY
c[Y/2,X/2] = FIRE               # Set site in the center to FIRE
```

The first statement uses Python/`numarray`-style multidimensional slicing to assign all states to *ready.* The second sets a single site in the center to firing.

One may also read out signal values using subscripts. For example,

```
a = c[0,5].value()
```

reads out the scalar integer value at coordinates $(0, 5)$. We must call the `value` method to read the actual value at `c[0,5]`. This is because subscripting a signal actually returns a `SignalRegion` object referring to a specific region of a signal, rather than the value itself. If one writes '`a = c[0,5]`', then `a` is a `SignalRegion`. To get the actual value, one must use `a.value()` to dereference the `SignalRegion`; conceptually, a `SignalRegion` is similar to an address (pointer) in a language like `C`. In contrast to reading values, is not necessary to explicitly dereference a `SignalRegion` when writing values using subscript assignment (as in '`c[:,:]  = READY`'), because

Python provides special hooks for handling subscript assignment[3].

Because a `SignalRegion` refers to an area, rather than a specific value, it can be used in a variety of more general contexts, such as making named neighbors and constructing `Read` and `Write` operations. Note, however, that inside of an event's transition-function, where only the value is of interest, one does not use the `value` method; this is because subscripting in that context automatically returns the value.

One can also use subscripting to read out arrays of values; for example, to read out the 5×5 array of values from $(0, 0)$ up through but not including $(5, 5)$ one, would use

```
    arr = c[0:5,0:5].value()
```

The array returned is a `numarray` object. One can also assign slices using array values

```
    c[15:20,10:15] = c[0:5,0:5].value()
```

Note that the `numarray` need not have come from a `Signal`; it could just as well have come from a file or have been constructed on-the-fly (as in `numarray.zeros((5,5))`).

To read the entire array, one can use a statement like

```
    arr = c[:,:].value()
```

Note that if the signal region refers to a single site, `value()` returns scalar element and if it refers to multiple sites, `value()` returns an array.

SIMP uses the `numarray` module extensively for representing and manipulating

[3]Subscript assignment only works when a subscript expression appears on the left-hand side of an assignment. Given a named `SignalRegion` obtained with code like '`a = c[0,5]`', one must instead write the value there using the assignment attribute as in `a._ = 2`.

`Signal` data. The `numarray` module and classes provide full Python support for multidimensional arrays. This support includes utilities for generating arrays, saving them to files, and performing all kinds of transformations and analyses.

A `numarray` object can be saved to a file in a variety of ways. One way is to use the Python `pickle` module.

```
import pickle
pickle.dump(arr,open("state_file","wb")) # save array to "state_file"
```

To read the file back, use

```
arr = pickle.load(open("state_file"))
```

Another option is to use the `tofile` method of the array. Certainly, there are many other ways as well. While we mention some of the useful `numarray` package features in this tutorial, one should see the Numarray documentation available at `http://www.stsci.edu/resources/software_hardware/numarray` for full details[4].

4.1.5 User interface

All of the example SIMP programs instantiate a `Console` object to perform on-screen rendering and provide an interactive user interface. In the current example, the code for declaring and initializing a `Console` is

[4]SIMP also defines some additional helper numarray functions such as `makedist`, `getdist`, `arraytopnm`, `magnify2d`;

```
ui = Console(tricolor_rend)    # Instantiate a console called "ui"
                               #   and initialize its renderer
ui.bind("DYNAMICS",gh_event)   # Bind the console's dynamics handler
ui.start()                     # Start the interactive interface
```

The first line creates a `Console` object called `ui` and brings up an on-screen viewer using `tricolor_rend`, the renderer previously defined. Because `tricolor_rend` implements the `Renderer` interface, the the `Console` knows how to use it to generate rendered image arrays and to control the region that is rendered.

The `Console` `bind` method is used to bind custom commands to key-press events. For example, to re-initialize the CA state when `S` is pressed,

```
def seed():
    "Initialize all sites to READY with a FIRING site in the center"
    c[:,:] = READY          # Set all sites to READY
    c[Y/2,X/2] = FIRE       # Set the center site to FIRE
ui.bind("S",seed)           # Bind 'seed' to key "S"
```

Lower-case keys are reserved for predefined commands; user-defined commands should be bound to upper case keys. Brief documentation derived from the documentation strings of bound commands is printed when the user presses the help key, `h`. For this reason, it is a good idea to define a documentation string as is done above.

In the code, '`ui.bind("DYNAMICS",gh_event)`', binds `gh_event` to the console's dynamics handler. A '`DYNAMICS`' event triggers an update of the dynamics. The console handles them by calling `gh_event` and invoking the `renderer` to display the result on-screen. Pressing `Space` generates a single '`STEP`' event, while pressing `Enter` causes them to be scheduled repeatedly until this behavior is turned off by pressing `Space`.

Finally, calling `ui.start()` starts the interactive interface, and does not return until the user quits by pressing `q`.

4.1.6 Scripting

One need not define an interactive user interface if the program is meant to run in script mode. For example, the following code runs several iterations and generates a sequence of portable pixmap ('.ppm') images like the ones shown in Fig. 3·2 (c)

```
for i in range(4):
    img_arr = tricolor_rend() # get rendered version of current state
    open("gh%i.ppm" % i,"wb").write(arraytopnm(img_arr)) # save to file
    gh_event() # do a step of the dynamics
```

Calling a renderer object returns the contents of the renderer's current view. Normally, arrays returned by a renderer are three-dimensional numarrays indexed by Y,X, and the color outputs. (If *grayscale* rendering is used, they will simply be two-dimensional arrays.) The SIMP helper function `arraytopnm` converts such arrays to '.ppm' formatted strings, which is a handy because the format is easy to read and write and can be converted to many other formats using Jef Poskanzer's widely available `netpbm` library and command-line tools.

One may also derive the `Console` from a script. For example, to display a sequence of four updates on-screen before starting the console, one could use the code

```
for i in range(4):
    ui.issue(" ")# issue a space character, triggers a single
                 # dynamics and a rendering operation
ui.start() # begin running the console
```

4.2 Stochastic Greenberg Hastings CA

As discussed in the introduction, we can randomize the excitable medium with the following event.

> Transition to *firing* with probability p if *ready* and a neighbor is *firing*; transition to *resting* with probability q if *firing*; and transition to *ready* with probability r if *resting*.

Implementing the new dynamics in SIMP amounts to adding stochastic transitions to the basic event. We'll create three signals—`P`, `Q`, and `R`—as binary random variables with the desired distribution—Pr(`P`=1)=p, Pr(`Q`=1)=q, and Pr(`R`=1)=r. We'll use these signals as 'unfair coin tosses' when deciding whether to take a transition—if a signal's value is `1` the transition will be taken, otherwise it will not. The new signal declarations[5] are

```
binary = SmallUInt(2) # Binary state variable type with values in {0,1}
P,Q,R = map(Signal,[binary]*3) # Three signals for randomness
```

The modified event is

```
def stochastic_gh():
   if   c==READY and P==1:
     if (c[-1,0]==FIRE or c[0, 1]==FIRE or
         c[ 1,0]==FIRE or c[0,-1]==FIRE):
       c._ = FIRE                      # Stochastic transition to FIRE
   elif c==FIRE and Q==1: c._ = REST    # Stochastic transition to REST
   elif c==REST and R==1: c._ = READY   # Stochastic transition to READY
```

[5]Note, we could just as easily have declared these signals as `P,Q,R = map(Signal,[SmallUInt(2)]*3)` but we chose to demonstrate the use of a type object.

We have not explained yet how `P`, `Q`, and `R` implement the desired random distributions. The most direct way is to use a random number generator to assign random values to the signals so as to fulfill the desired distribution. SIMP provides `makedist` to do this. The function takes two arguments—the shape of the output array and the distribution of values. For example, within the `set_parameters` function of the `stochastic_greenberg_hastings.py` code, the line

```
  P[:,:]  = makedist(P.shape,[1-p, p])
```

assigns the values of `P` to a newly generated random array with the same shape as `P` and having values of 0 and 1 distributed independently with probability $(1-p)$ and p; similar code sets the distributions for `Q` and `R`. (Note that distributions need not be normalized to one—a distribution parameter of `[3,4]` would create an array with a 3 to 4 ratio of zeros to ones.)

Although this strategy of generating distributions works, it requires an expensive call to a random generator *for each array element*. Regenerating distributions in this way before *each step* would slow computations considerably. Fortunately, there is a less expensive way. Because our event can not 'see' long-range correlations—local information tends not to travel too far before 'diffusing'—we can *regenerate* the local random variables by *stirring* them. By rearranging the same data in a non-local way—say, by shifting a `Signal` by a random amount[6]—we can cheaply "recharge" the patch of randomness that a locale sees. It's like a shell game in which a sequence of small patches from a much larger space are revealed randomly, and unless the dynamics is an especially 'smart' adversary tuned to our game it will not be able to tell that the patches it is shown come from our cheaper source of stirred randomness.

[6]This is the policy that the current STEP implementations employ under the hood. The ideal stir operation is a random permutation, however, a true random permutation is expensive to implement. In the future STEP implementations may make more sophisticated, but still inexpensive, implementations of `Stir` available that come close to a random permutation.

To stir the distribution before each update, we use the `Stir` operation and create an update `Sequence` that stirs the random signals before calling the stochastic event.

```
stochastic_gh_step = Sequence([Stir([P,Q,R]),
                               Event(stochastic_gh)])
```

We have introduced two new STEP operations—`Stir`, which we just explained and `Sequence`. A `Sequence` is not really an operation in itself, but an ordered sequence of STEP operations. In addition to being a useful programming construct, a `Sequence` informs the STEP about sequences of operations that will be called together. A STEP may then optimize such a sequence.

Finally, we outline the methods used to obtain the data plotted in Fig. 3·3. The data was gathered by running an outer loop that iterated over the p values in increments of .01 from 0 to 1. For each value, ten trials were run. At the beginning of each set of trials, a randomized assignment was used to load a new distribution corresponding to the new value of p into `P`. At the beginning of an individual trial, `c` was initialized to all *ready* except for a single *firing* spark in the center. Next, an inner loop repeatedly invoked `stochastic_gh_step()`, checking every tenth step to determine whether 'the fire had burned out' by examining the distribution of states returned by '`dist = getdist(c[:,:].value(),0,3)`'. `getdist` is a SIMP helper function that, given an array and a range of integer values, returns a histogram vector giving, for all values in the range 0–3, the number of sites having that value. Once it was determined that the fire had burned out, the fraction burned was computed from `dist`.

4.3 A one-dimensional CA example

One-dimensional systems are usually rendered using space-time diagrams in which the history of several consecutive states is displayed as a two-dimensional image, with the horizontal axis representing space and the vertical representing time. The `XTRenderer` provides special support for space-time rendering. We present a simple CA called Parity as an example. We define it on a one-dimensional lattice with binary signals. The transition-function adds the left, right, and center values and yields 1 if the sum is odd or 0 if the sum is even.

4.3.1 The program

The code for the Parity dynamics is

```
from simp import *
X=50
initialize(size=[X])    # 1D grid
# -------------------------------- SIGNAL DECLARATION
c = Signal(SmallUInt(2))  # binary state
# -------------------------------- TRANSITION FUNCTION
l,r = c[-1],c[1]  # declare the neighbor directions
def parity():
    c._ = l^c^r  # ^ denotes 'xor', sets bit if 'l+c+r' is odd.
parity = Event(parity)
```

The code declares `l` and `r` to represent `c[-1]` and `c[1]` (left and right)—although `c[-1]` and `c[1]` could instead have been used in the transition-function we wanted to demonstrate the use of named `SignalRegion` objects. The code below declares a space-time renderer, initializes the state to zero with a single one point in the middle, and creates a console.

```
# -------------------------------- RENDERING
white = IoSignal(UInt8)
def bw():
    if not c: white._ = 255
bw_xt = XTRenderer(Event(bw),white,time=X/2)


# -------------------------------- INITIALIZE
c[X/2]=1  # point seed in the center.
# -------------------------------- CONSOLE
ui = Console(bw_xt,mag=8) # set renderer and default magnification
ui.bind('STEP',parity) # Specifies the 1D renderer.
ui.start()
```

Rather than employing the usual renderer, we employ a `XTRenderer` to capture the space-time history of the CA. An `XTRenderer` object defines a special method called `record` for capturing the space-time diagram over a sequence of time steps. Every time the `Console` does a `STEP`, it looks for and automatically calls a renderer's `record` method if it's defined. The image in Fig. 4·1 depicts a space-time diagram generated by the console. Space runs horizontally while time runs vertically. The dynamics was initalized with a single value of '1' in the center and updated 24 times. The window of time recorded by the `XTRenderer` is set with the `time` parameter. In this diagram, time increases downwards. By using a negative value for `time` one can make time increase going upwards.

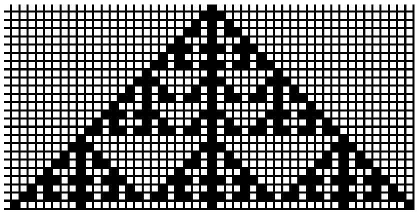


Figure 4·1: 1D Parity CA space-time diagram

In the example, the magnification was set to 8 using the `mag` parameter of the `Console`. (Grid lines were drawn because the magnification was high; to turn these lines off, set the `showgrid` `Console` constructor parameter to zero.) The image was collected interactively using the `CaptureView` command (`c`), but could have been captured using the script discussed below.

4.3.2 Using a script to record the history

Instead of using the console, one might capture the image of Fig. 4·1 using a script

```
bw_xt.record()   # Record the initial state
for i in xrange(24): # Do 24 updates
   parity() # call the event (does a step of the dynamics)
   bw_xt.record() # record the state
arr = bw_xt()  # get the output array
rescaled_arr = magnify2d(arr,mag=8,grid=1) # magnify with grid lines
open("out.ppm","wb").write(arraytopnm(rescaled_arr)) # save image
```

First it records the initial state, then it runs the dynamics 24 times, calling the `record` method of the `XTRenderer` after each update. Next, using `magnify2d`, it gets the output array, magnifies it, and adds grid lines. Finally, it converts the array to a '.ppm' string and saves it to the file, 'out.ppm'.

4.4 Programming HPP

Programming LGA on a rectangular grid is easy with SIMP. If we don't care about having non-interacting, interlaced dynamics, we can just specify HPP as follows:

```
binary = SmallUInt(2)
# allocate 4 state-variables
a = Signal(binary); b = Signal(binary)
c = Signal(binary); d = Signal(binary)
```

```
def hpp():
  if (a and c) and not (b or d): # only a and c collide
    b._ = 1; d._ = 1
    a._ = 0; c._ = 0
  if (b and d) and not (a or c): # only b and d collide
    a._ = 1; c._ = 1
    b._ = 0; d._ = 0


hpp = Event(hpp)

hpp_step = Sequence([
        Shift({a:[ 0,1], b:[-1, 0],
               c:[ 0,-1],d:[1,0]}),
        hpp])
```

The only new construct is a shift, which, for convenience is given as a Python mapping (dictionary) from the signals to their respective shifts.

4.5 HPP programmed on a sublattice

When one wants to program on a sublattice, namely the one shown in Fig. 4·3 things are a bit more complicated, namely one must describe the lattice via a generator matrix, and allocate state variables on it.

The lattice that we would like is the one defined by the shaded rectangles in Fig. 4·2. As depicted by the arrows, the generator vectors for this are $(0, 2)$ and $(1, 1)$. Note that the generator vectors induce the rectangular unit cell that's outlined in black.

We stack these generator vectors into a generator matrix and use this to redeclare the state variables as in

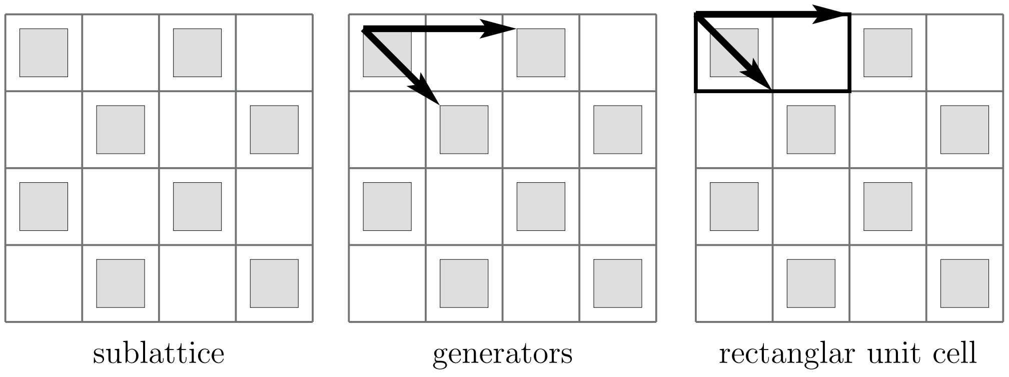


Figure 4·2: Checkerboard sublattice, its generators, and rectangular unit cell

```
gen = [[1,1],
       [0,2]]
a = Signal(binary,gen); b = Signal(binary,gen)
c = Signal(binary,gen); d = Signal(binary,gen)
```

Once this has been done, one is effectively programming on a sublattice. The only other thing that must be done is that the Event must be given on the same lattice and should be shifted together with the state variables:

```
hpp = Event(hpp,gen)

hpp_step = Sequence([
        Shift({a:[ 0,1], b:[-1, 0],
               c:[ 0,-1],d:[1,0]
               hpp:[0,1]}),
        hpp])
```

Rather than redundantly specify the generator everywhere, one can alternately leverage SIMP's facility for overriding the default generator (the grid) in `initialize` routine.

4.5.1 Cosets

At even times the state variables and event will be on the even coset and at odd times on the odd coset. If the program wants to know which coset an event or a state variable is on it can call the `get_coset` method as in ‘`hpp.get_coset()`’. Underneath, this calls a STEP `GetCoset` operation. The implementation will return a canonical representation based upon the grid coordinate of the site in the *rectangular unit cell*. Specifically, as Fig. 4·3 demonstrates, the coset representative is the site that falls within the lattice allocation’s rectangular unit cell. So, the ‘even-site’ coset is represented by the coordinate $(0, 0)$ while the ‘odd-site’ coset is represented by the coordinate $(0, 1)$.

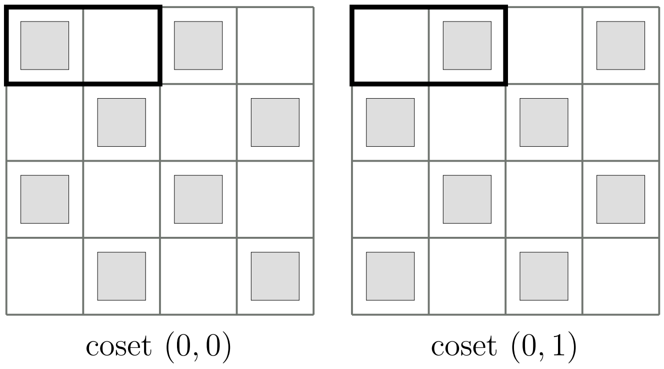
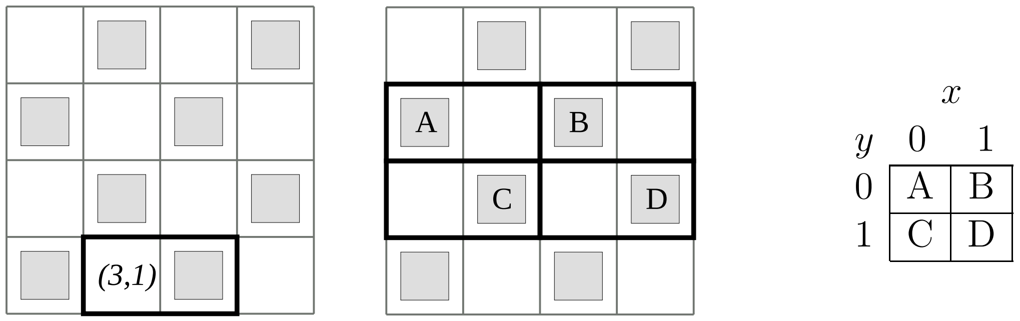


Figure 4·3: Coset representative coordinates come from location of site in unit cell

4.5.2 Accessing sublattice data

If one wants to access state variable values, this can be easily done even though the state variables don’t actually exist at all sites of the grid. This is because indexing is done in multiples of the sublattice’s rectangular unit cell. In particular, consider reading the state variable a at index $(3, 1)$ when the grid is on the odd coset. Even though the coordinate itself does not hit a state variable, because indexing is done in multiples of the rectangular unit cell, data in the site inside the box depicted in

Fig. 4·4 (a) is actually selected.



(a) `a[3,1]` (b) `a[1:3,0:4]` (c) `a[1:3,0:4].value()`

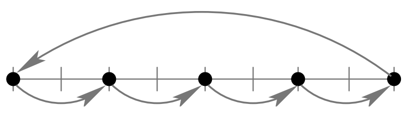
Figure 4·4: Selecting sublattice sites with the rectangular unit cell

If one wants to read out an array of state variable values, this may still be done by selecting a region. For example, selecting `a[1:3,0:4]` in Python selects the region defined by the points $\{\mathtt{x} : (1, 0) \leq \mathtt{x} < (3, 4)\}$. If the state variable had an allocation that was as dense as the grid, then the resulting array would be of size $(2, 4) = [(3, 4) - (1, 0)]$. But because its sparse, with a rectangular *spacing* of $(1, 2)$, the actual size of the resulting densely packed array is $(2, 2)$. The actual sites selected by this region are shown in Fig. 4·4 (b). If the user were to read the array selected by this region, (c) shows how the sites labeled in (b) would be mapped, by the rectangular blocks of the unit cell, into a dense 2×2 array.

Selecting a slice with an extent that's not a multiple of the rectangular unit cell yields a SIMP error. Because selecting a region that's not a multiple of the rectangular unit cell would yield a region of sites that would not map into a dense, rectangular array, doing this is not allowed. There are a number of problems with regions that aren't a multiple of the rectangular unit cell. For one, rows of such a region would not necessarily all contain the same number of sites, and in order to get a dense packing into an array, a 'jagged' array with different lengths depending on the row would be needed.

4.5.3 Wrap-around

There's a related problem when it comes to wrapping a sublattice on a torus—in particular, the sublattice must wrap around to itself and not some other coset. Consider Fig. 4·5, which illustrates the problem in one dimension. If one chooses an odd-number size for the dimensions as in Fig. 4·5 (b) the lattice wraps around to a coset other than the one it started on. The result is that it's not really the same lattice anymore, rather, its actually the interleaved lattice that we originally tried to escape[7]. In order to avoid this, and the unnecessary implementation complexity that would be associated with this situation, SIMP raises an error if one tries to create a lattice allocation that's not compatible with the size of the grid.


(a) compatible grid and sublattice

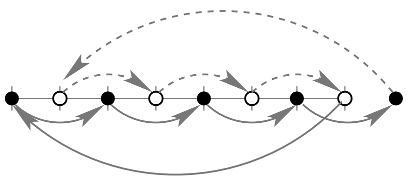

(b) incompatible grid and sublattice

Figure 4·5: Compatible and incompatible grid sizes

Conceptually, the size of the grid is incompatible if it is not a multiple of another rectangle induced by the lattice, namely the least common orthogonal unit cell. This other unit cell is the smallest rectangular unit cell on which the lattice repeats under a rectangular tiling. Fig. 4·6 outlines this unit cell for HPP's sublattice. When a user selects a lattice that does not wrap, it suggests a boundary size that will wrap by rounding up to the next multiple of the least common orthogonal unit cell.

[7]In the event that such an interleaved lattice is desired, one can program the dynamics in a form that has two locally non-communicating sublattices, and then use an odd size to make what amounts to the topological equivalent of a M obius band where the locally non-communicating sets of odd and even sites are actually on the same lattice as far as global interaction is concerned and correspond to opposite "faces" of the band.

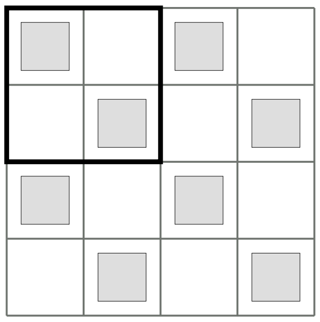


Figure 4·6: least common rectangular unit cell

4.5.4 Rendering options

Unlike reading array values, where one would actually like to achieve a dense packing of state variables into the array, in rendering for on-screen display, one would like to preserve, as much as possible, the logical geometry of the grid coordinates. To achieve this, one can create a rendering update that writes to color output channels that are defined on the grid. This comes at the cost of adding extra pixels for all sites of the grid, rather than just the ones where the state variables are, but has the benefit of preserving the logical geometry. In the examples that follow, we'll be rendering HPP's state variables as a gray-scale sum in two slightly different ways.

Fig. 4·7

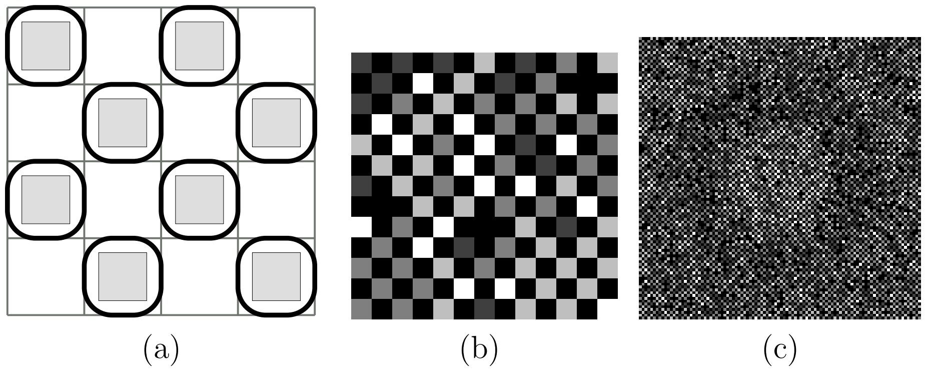


Figure 4·7: Rendering HPP onto a sparse subgrid

Fig. 4·7 shows the first example that we'll consider. In (a) the rounded, black boxes mark the grid sites where we'll render the state variables. The grid sites on the odd coset will remain blank (black). To define this kind of rendering event, first we allocate a IoSignal on the grid. The grid has the orthonormal generators $(1, 0)$ and $(0, 1)$, so the generator matrix is $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$.

```
orth     = [[1,0],
            [0,1]]

sub      = [[1,1],
            [0,2]]

white = IoSignal(UInt8,orth)
```

The rendering event uses a grayscale sum to represent the number of particles. It outputs to the site where the event takes place, but not to the neighboring site. There, it writes zero.

```
def intensity():
    white[0,0]._ = (a+b+c+d)*63
    white[0,1]._ = 0
```

Of course, the rendering event itself must be defined on the sublattice[8].

```
render_event = Event(intensity,sub)
rend = Renderer(render_event,outputs=white)
```

The last thing that needs to be done is to make the rendering event's coset follow

[8]One might be tempted to define a rendering event on the orthonormal lattice of the grid and use that to read state variables that are defined on a sublattice. This, however, is an illegal STEP event because it operates on a state variable defined on a coarser grained lattice and thus is not uniform—it's behavior depends on the coset with respect to the state variables.

that of the `hpp` event. This can be done via the `SetCoset` operation as follows.

```
dynamics = Sequence([dynamics,
                     SetCoset({render_event:hpp})])
```

`SetCoset` sets the coset of a lattice allocation to either a static location or one defined by another lattice allocation. Used in the latter way, its a useful method of getting one lattice allocation to 'follow' another.

Another strategy for rendering is, rather than leaving one coset empty, to fill its values as well. In Fig. 4·8 we render the sublattice state variables in horizontally oriented tiles that have the shape of a brick wall. This can be accomplished by just changing the `intensity` event function to

```
def intensity():
    white[0,1]._ = white[0,0]._ = (a+b+c+d)*63
```

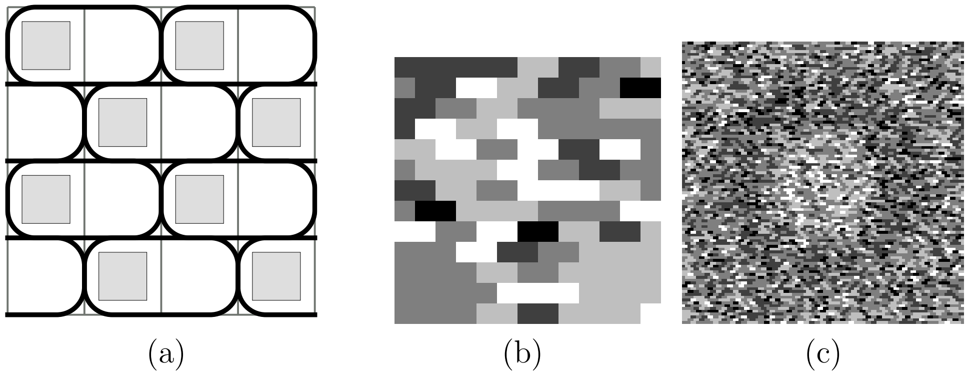


Figure 4·8: Rendering onto 'brick wall' tiles

4.5.5 Programming HPP by packing sublattice state into an array

Before it was possible to program HPP using sublattice constructs, one was required to resort to more manual techniques to pack the structure into an orthogonal array storage format. In particular, to implement HPP, it was necessary to pack the state into an orthogonal array and modify the local and/or global topology of the problem coordinates to accommodate this remapping. We present two array-packing strategies to demonstrate the improvement that structures like sub-lattice allocations add. In order to fully appreciate the simplifying nature of the sublattice constructs it is helpful to see these alternate, more complicated strategies. In anticipation of the implementation techniques discussed in Chapter 7 we also present these strategies as a concrete example of what the STEP implementations automatically abstract for the user.

In order to implement HPP, as was necessary before we created the sublattice extensions, there are two fundamental issues one must resolve. The first is how to represent and store state-variables that, in the logical model are on a sparse, non-covering sublattice of the n-dimensional integer grid. The second is, given a storage strategy, how to map logical indexes and coordinates into physical indexes. We present two variations of a technique for doing this. Both variants are based upon tessellations of the lattice's rectangular unit cell and map the logical topology of the space to an altered, less-intuitive physical memory layout.

We call the two different mappings the *orthogonal tessellation mapping* and the *lattice-aligned tessellation mapping*. The orthogonal tessellation maps sublattice points to array locations associated with unit cells of an orthogonal projection of the lattice. Fig. 4·9 depicts this strategy graphically. In the figure, the letters in the middle of the figure mark the logical lattice sites and on the right mark their storage locations in an orthogonal array.

The lattice-aligned tessellation maps sublattice/coset points to array locations associated with a lattice-aligned tessellation of the rectangular unit cell. Fig. 4·10 depicts this strategy in a fashion similar to Fig. 4·9.

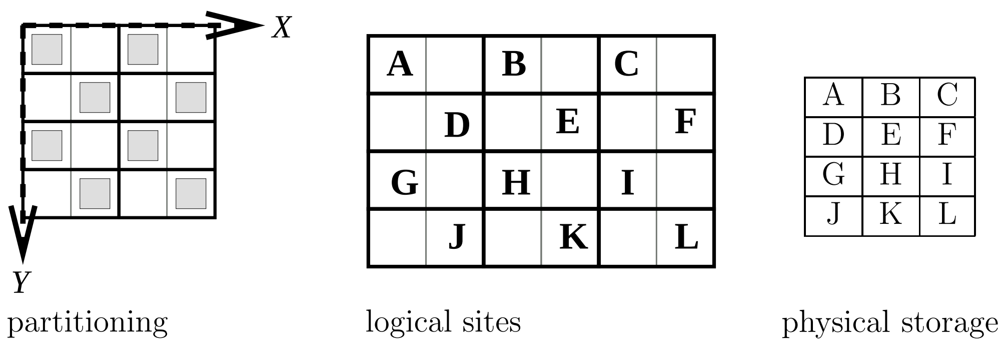


Figure 4·9: Orthogonal tessellation for mapping of the checkerboard sublattice into an orthogonal array.

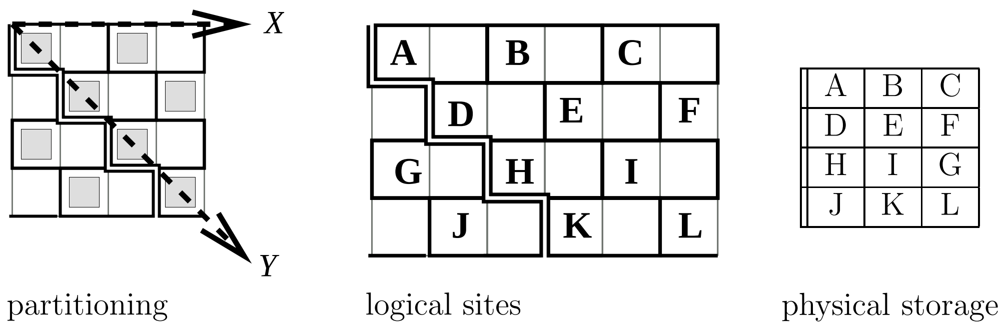


Figure 4·10: Lattice-aligned tessellation for mapping of the checkerboard sublattice into an orthogonal array.

Fig. 4·11 and Fig. 4·12 demonstrate what happens when we remap local shifts in the orthogonal and lattice-aligned tessellations into array indexes.

Table 4.1 summarizes the re-mapping from the logical shifts to the physical shifts. Notice that the adjustment to the shifts in the orthogonal packing depends on both space and time—successive HPP updates alternate between the odd and even cosets of the lattice. The lattice-aligned mapping depends only on time. The result is that the scan loop for the lattice-aligned strategy is uniform and can be implemented by a spatially-uniform LGA. However, this does have a cost. In particular, the lattice-aligned mapping has a global skew that's not present in the orthogonal mapping. This can be seen graphically in Fig. 4·13, which contrasts two configurations of HPP after 48 updates starting with an initial square vacuum in the center. The dynamics

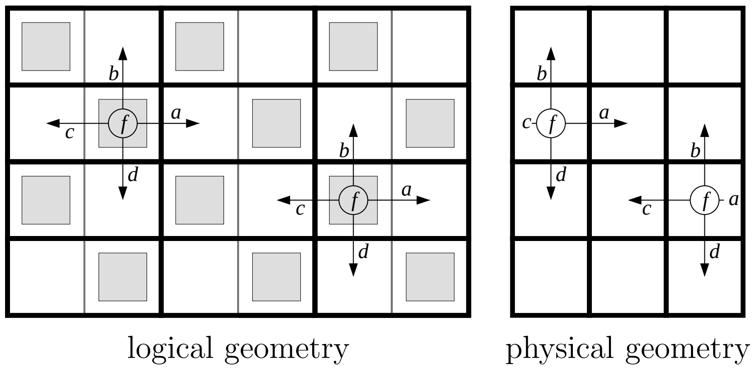


Figure 4·11: Shifts in an orthogonal packing

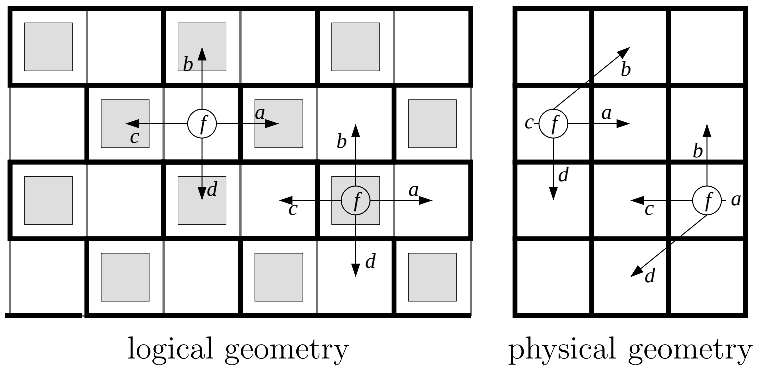


Figure 4·12: Shifts in an lattice-aligned packing

is the same—there is a one-to-one correspondence between sites—but the geometry is skewed. This is detremential not only to renderning, but also to the the indexing and reading of values. It also makes coupling the dynamics of multiple systems much more difficult—if their implementation requires different mapping techniques, then the programmer had better find a unified technique that works for all.

The skew also has an effect on the global boundary. Usually one wants to have a torus that wraps around without a shear at the boundary. In this case it is necessary to employ the orthogonal mapping. However, this bears the cost of a scan that has spatial non-uniformity. (As such, it is not directly implementable in the CA paradigm, so the usual strategy is to employ the lattice-aligned mapping with its skew. As discussed in Section 7.1.2, the STEP implementations actually internally

	logical shift	physical shift			
		orthogonal		lattice-aligned	
signal		$y+t$ even	$y+t$ odd	t even	t odd
a	$(0,1)$	$(0,0)$	$(0,1)$	$(0,0)$	$(0,1)$
b	$(-1,0)$	$(-1,0)$	$(-1,0)$	$(-1,0)$	$(-1,1)$
c	$(0,-1)$	$(0,-1)$	$(0,0)$	$(0,-1)$	$(0,0)$
d	$(1,0)$	$(1,0)$	$(1,0)$	$(1,-1)$	$(1,0)$

Table 4.1: Mapping HPP's logical shifts to physical values

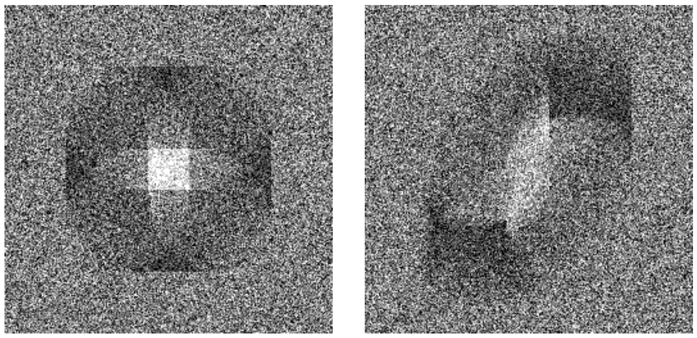


Figure 4·13: The effect of the lattice-aligned skew on HPP

employ the orthogonal mapping.)

Below, we provide the extra SIMP code—altered shifts and update steps—that are necessary program HPP with a lattice-aligned packed array.

```
#-------------------------------- Declare the update
hpp_step0 = Sequence([
        Shift({a:[ 0, 0],b:[ -1,0],
               c:[0,-1],d:[1,-1]}),
        hpp])

hpp_step1 = Sequence([
        Shift({a:[0,1],b:[-1,1],
               c:[0, 0],d:[1,0]}),
        hpp])

hpp_step = Sequence([hpp_step0,hpp_step1])
```

Notice that is longer and less intuitive than the sublattice extensions presented in Section 4.5. The bottom line of this discussion is that getting the indexes right is an error-prone process and something that, in the general case of packing lattices into orthogonal arrays, is better left to a software compiler. STEP supports exactly this. As Section 8.1.3 discusses, this is a major advantage that STEP has over other data-parallel array languages.

4.5.6 Programming HPP as a Margolus neighborhood BPCA

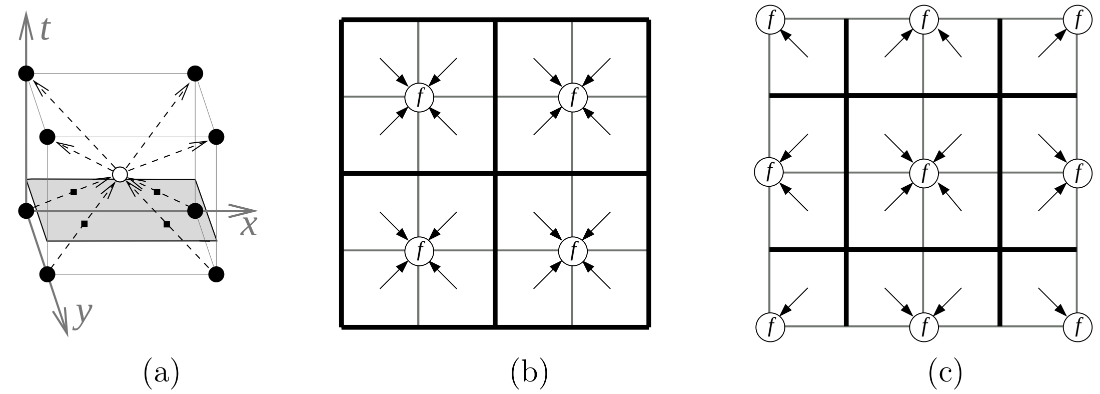


Figure 4·14: Margolus 'neighborhood' partitioning CA

It turns out that there is *yet another* way to prgram the HPP lattice-gas.

Now that we've introduced the relevant constructs, it's fairly easy to re-program HPP as Margolus neighborhood BPCA. The Margolus neighborhood presentation of the HPP lattice gas from (Toffoli and Margolus, 1987) represents one of the more popular ways of programming the HPP lattice gas.

In this approach the four distinct LGA signals that impinge on an event are mapped into a single signal in 2×2 blocks. These blocks are defined by the cosets of an update event that updates the signals in blocks of four. As Fig. 4·14 (a) shows, the block partitioning can be derived by slicing the space-time structure mid-way between two events. The in-flight positions of the signals become their new positions in the cosets of the finer-grained grid and the positons of the events define the block partitionings. Fig. 4·14 (b) shows the two block partitionings.

The Margolus neighborhood HPP dynamics updates sites in blocks of four with an function that, by default, swaps block contents blocks horizontally. However, if exactly two particles collide head-on, it swaps the contents vertically. Fig. 4·15 depicts the transition funtion and while Fig. 4·16 presents an example of the dynamics on the right. The shifting of the partitionings causes particles to move diagonally with inertia defined by the coset position with respect to the partitioning.

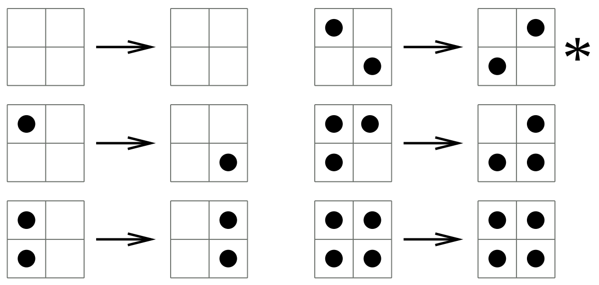


Figure 4·15: Margolus Neighborhood local transition function

Now, in order to implement the HPP lattice gas in SIMP, we'll need particles implemented by a binary state variable defined on the grid and an event whose

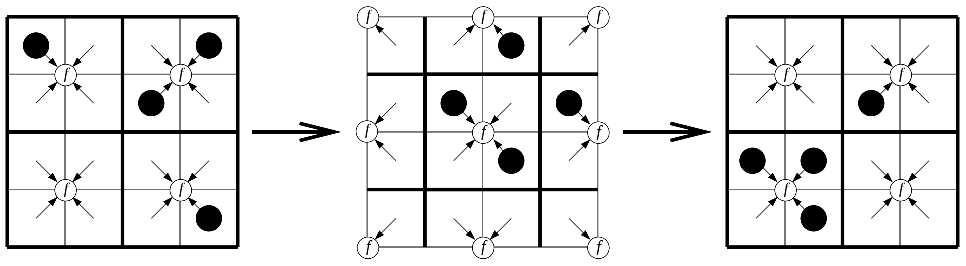


Figure 4·16: Margolus Neighborhood HPP example

lattice has a spacing of 2 in both the X and Y directions. In between updates, we'll need to shift the event. Fig. 4·17 (a) depicts the lattice for the event and its generators. Fig. 4·17 b and c show the two cosets that we'll want the event lattice to alternate between. The dark lines indicate the partitioning that the event will induce with its inputs and outputs.

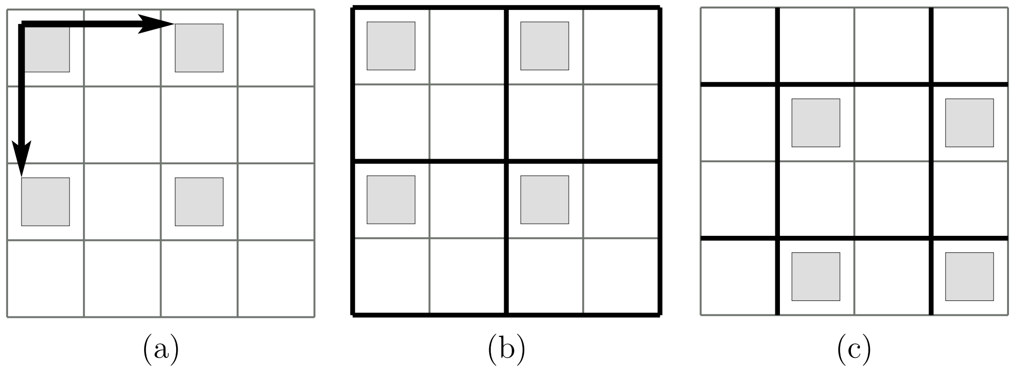


Figure 4·17: Margolus neighborhood event lattice and the partitions it induces

The lattice for the events is given by the generators $[0, 2]$ and $[2, 0]$. As is obvious from Fig. 4·17, the event lattice shifts by $[1, 1]$. Note that, because an event has no data, shifting the lattice of an event is always just a matter of updating a logical offset. Only the coset value of the offset matters.

```
from simp import *
initialize(size=[100,100])      # grid size
p  = Signal(SmallUInt(2))   # signal allocated on the grid
hpp_mesh = [[2,0]     # coarse-grained mesh for the Events
            [0,2]]    # spacing of (2,2)
def hpp():
    if ( (p[0,0]==p[1,1]) and (p[0,1]==p[1,0]) ): # head-on collision
        p[0,0]._ = p[0,1]; p[0,1]._ = p[1,1]      # (rotate 90 deg)
        p[1,0]._ = p[0,0]; p[1,1]._ = p[1,0]
    else:                                         # keep moving
        p[0,0]._ = p[1,1]; p[0,1]._ = p[1,0]      # (swap diagonally)
        p[1,0]._ = p[0,1]; p[1,1]._ = p[0,0]

hpp_event = Event(hpp,generator=hpp_mesh) # Declare Event on sublat
dynamics = Sequence(hpp_event,                # Apply the event and
                   Shift({hpp_event:[1,1]}) ) # shift to next block
```

4.6 Programming the polymer BPCA

We come to our final programming example, the Polymer dynamics. As discussed in Section 3.4, the polymer dynamics is a stochastic dynamics where polymer particles move randomly so long as they don't break a link. In order to guarantee that this constraint holds, we program the polymer dynamics on a sublattice.

We employ the Event stencil given in Fig. 3·14 to update the polymers. The stencil has a generators of $(2, 2)$, Therefore, we have a generator of $\begin{bmatrix} 2 & 2 \\ 0 & 4 \end{bmatrix}$. In programming the dynamics we introduce a random bit for deciding which of the two possible diagonal swaps the dynamics will try. Because it is only needed for the events, we allocate this state variable on the same lattice as the events. The for representing the polymer, we create a state variable on the grid.

```
# -------------------------------- STATE
mesh    = [[2,2], # generator for Events and the random bit
           [0,4]]

p      = Signal(SmallUInt(2))         # polymer state
r      = Signal(SmallUInt(2),mesh)    # random bit for emulating
                                      # Poisson updating
```

The dynamics is expressed as a event that tries to swap according to one diagonal or the other depending on the value of the random bit. It decides whether it's feasible to swap along the given diagonal by looking at the sites, if there's a swap, may create or break a link. If none of these sites has a polymer, then the swap may proceed.

```
# -------------------------------- DYNAMICS
def thermalize():
    """Swap if doing so doesn't create or break a polymer chain."""
    if r: # emulate Poisson updating with the random bit.
        if ((p[-1,0]+p[0,-1]+p[2,1]+p[1,2])==0):
            p[0,0]._ = p[1,1]  # swap along the '\' diagonal
            p[1,1]._ = p[0,0]
    else:
        if ((p[1,-1]+p[2,0]+p[-1,1]+p[0,2])==0):
            p[1,0]._ = p[0,1]  # swap along the '/' diagonal
            p[0,1]._ = p[1,0]
```

To provide more randomness, we choose to change the coset that the event applies to randomly. In particular, we 'stir' the event. Absent any data to stir, stirring a event just randomizes its coset. This achieves the desired effect of evenly moving the center of the stencil uniformly to all of the sites.

```
thermalize_event =  Event(thermalize,generator=mesh)
thermalize_step = Sequence([thermalize_event,
                            Stir([r,thermalize_event])])
```

Chapter 5

The STEP API and Architecture

The STEP API is an interface to the capabilities of an abstract cellular automata machine called a STEP, which short for *space-time event processor.*[1] With the STEP API an application may employ a STEP as a co-processor for performing CA computation.

The preceding chapter on the SIMP programming environment implicitly introduced the STEP CA co-processor model that the STEP API encapsulates and some of the constructs that API provides. In particular, a STEP is a machine with facilities for allocating spatially distributed state variables and issuing parallel operations on them. This chapter explains in detail both STEP's logical computation model and its API.

The logical model

The STEP logical model is that of a space-time event processor; its computations are defined by a *spatial topology* and temporal sequence of *computations.* As a prerequisite for explaining STEP's computational model we first present its spatial model in Section 5.1. A novel feature of the model is that it allows one to declare state variables and operations on sublattices. We spend some time on the implications of this.

[1]Our abstract STEP architecture and API has its origins in the STEP interface to the CAM-8 cellular automata machine. This interface served as a starting point for our work and some aspects of our API bear some resemblance to it. However, we have attempted to make our API more machine independent, a bit higher-level, and incorporate new concepts like sublattices.

Having defined the geometric model, we detail the computational model in Section 5.2. In particular, we give a mathematical definition of the `Signal` (state variable) construct and the operations that act upon them, including the following

- `Event`—applies a transition function to signal objects in parallel,
- `Shift` —translates signals (or events) by a given amount,
- `Read/Write`— I/O operations that read and write a regions of state variables into and out of arrays,
- `Stir`—re-arranges the values of a state variable in a non-local, way.[2]

The API

After defining the logical aspects of the STEP model, Section 5.3 turns to practical matters of representing the logical computations with a programming interface. The interface defines a run-time model for controlling a STEP as a co-processor and the code structures that encapsulates it.

STEP's notion of the role of a CA processor is similar to that of a video card; *a STEP is more of a co-processor than a full-fledged processor in its own right.* In particular we have chosen to base the API on a model that, like the video-card/api/application model of OpenGl (Segal and Akeley, 1994; Segal and Akeley, 2001),

- provides a means for declaring and issuing operations, but relegates control decisions to a controlling program run on a general-purpose processor,
- maintains as little internal state as possible in order to make the implementation simple, and

[2]This is a "poor man's permutation" which rearranges a state variable in an arbitrary way. At the very least it applies a large shift, at the best it performs a more significant permutation. The exact behavior is implementation dependent.

- does not include control structures like "if" statements, but instead delegates control tasks to the application that invokes the API.

Because a STEP can operate more efficiently if sequences of operations can be issues as a batch, we provide a *sequence* operation for declaring and issuing batch sequences operations. Because one would not like to create a control bottleneck in by having a STEP have to wait for instructions, the API is fundamentally asynchronous, but has methods, callbacks, and a blocking flush primitive that can be used to synchronize operations.

The run-time model is simple. First one instantiates a `Step` class, defining the topology of its space at that time. Then one declares some state variables and operations that act on them. Finally, one issues operations causing the STEP to perform them.

In its implementation, the API consists of a simple initialize/declare/issue interface to the processor itself and a set of data structures for representing state variable allocations and STEP instructions. The processor interface is that of sparse a class consisting primarily of (1) a constructor for passing *initialization* arguments such as the size of the grid and other optional arguments to a STEP, (2) a method for *declaring* state variables and operations, and (3) a method for *issuing* operations.

5.1 The STEP geometric model

In our present work, we postpone the thorny issue of how best to specify the global topology of the CA space. Instead, we focus on boundaries that wrap-around as a torus. The beauty of a toroidal boundary is that, really, there is no boundary. In this way a torus approximates infinite space that has no boundaries. If a fixed-value boundary is needed, one may fabricate it within a CA by defining a read-only pattern that indicates indicates where the boundary is and a dynamics that respects it.

5.1.1 Global topology and the grid

A STEP has a local topology that's defined by a discrete, n-dimensional, orthonormal integer lattice in Cartesian space. We call this lattice the *grid*. The sites of the grid are given by the integer vectors x in $\mathbb{Z}^n$. Coordinates x are are given as vectors in $[z, y, x]$ form—with the most significant dimension first.[3] Fig. 5·1 (a) depicts the coordinates grid.

The integer grid is the finest granularity of coordinate representable in STEP. We choose this topology because it by far is the most common topology in CA models. As a practical matter, structures embedded in this the grid topology map conveniently onto computer memory and computer display indexing schemes. This stands in contrast to more generic topologies such as that of the real numbers.

The overall topology is that of a grid that wraps around as a torus. Fig. 5·1 (a) shows a grid with *size* (6,6) and labels each site with its coordinates. To emulate a boundary-less space, the coordinates wrap-around with the bottom wrapping to the top (b) and the left wrapping to the right (c) yielding a torus.

The global topology is that of a finite torus whose size is given by a positive dimension size vector d. Coordinates are taken modulo d (x_i equivalent to $x_i + kd_i$ for any integer k).

We choose the global topology of an n-dimensional torus because 1) it is a common choice for CA models, 2) it approximates a boundary-less space, and 3) is easy to

[3]This reflects the positional notation implicit in Arabic numerals where the number one hundred and twenty three is written from left to right in decreasing order of significance as 123—more significant digits are appended on the left. It also reflects the storage conventions of `C` and `numarray` where indices of lower significance are stored closer together. When `C` and `numarray` map a multidimensional array to a 1D memory array, the unit-stride dimension—the one with the highest degree of locality—is the least significant, right-most one. When there is a choice, the programmer should align neighbor access with lower dimensions, because, depending on the STEP implementation, doing so increases data locality and may make the computation more efficient. In accordance with computer graphics and typographic conventions, for rendering and display purposes, X goes to the right, Y goes down and Z goes 'behind'.

describe. Other types of boundary One might reasonably extend the architecture's definition to handle other types of boundary conditions. However, because it is not clear which other boundary types should be supported or how they should be represented and it is possible to emulate other types of boundaries such as fixed boundaries inside of STEP by appropriately programming the CA dynamics with an extra constant signal that indicates where the boundaries occur, we postpone this matter, we postpone this issue for future work.

The number of dimensions and size of the torus is declared when a STEP is initialized through a `size` argument to the STEP class constructor.

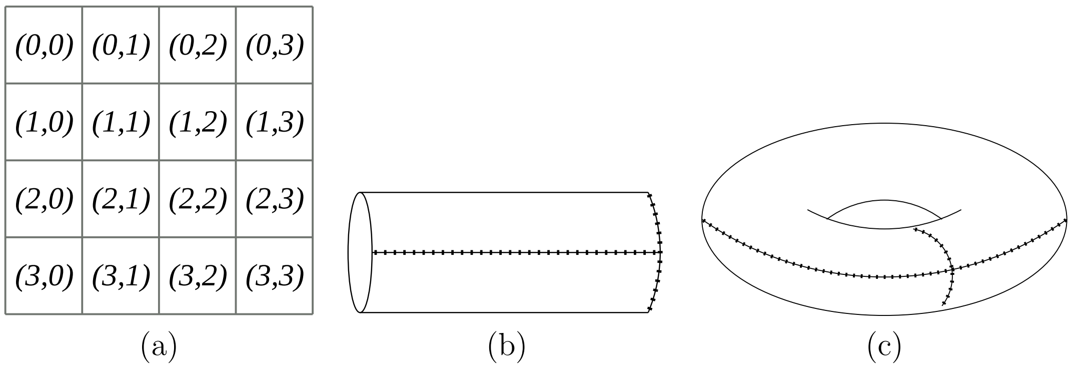


Figure 5·1: The grid and wrap-around

5.1.2 Lattices

In the later sections of Chapter 4.4 we introduced the notion of a lattice, sublattice, cosets, the grid, the rectangular unit cell, and sublattices. We now define these concepts in more detail.

We start with the concept of a lattice. A *lattice* is a set of regularly-spaced points. The points are called *sites*. A lattice is generated by linear combinations with integral coefficients of a set of *generator vectors*. Starting at any lattice site, integer combinations of the generator vectors generate the rest of the sites that compose the lattice.

An n-dimensional lattice has n generator vectors. Mathematically speaking, an n-dimensional lattice Λ is defined as the set of points generated by multiplying n generator vectors g_i, $0 \le i < n$ by integers a_i;

$$\Lambda = \{\sum_i^n a_i \mathsf{g}_i : a_i \in \mathbb{Z}\},$$

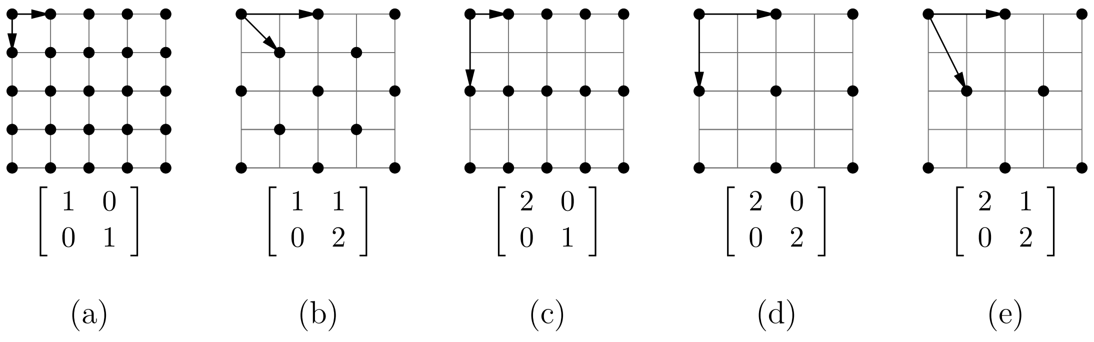
More conveniently, one may express the set of generator vectors as a matrix G

$$\mathsf{G} = \begin{bmatrix} \mathsf{g}_0 \\ \mathsf{g}_1 \end{bmatrix}$$

and express a lattice in terms of the generator matrix as follows

$$\Lambda = \{\mathsf{aG} : \mathsf{a} \in \mathbb{Z}^n\}.$$

Fig. 5·2 depicts five different 2D lattices atop their respective generator matrices. The circles denote the sites of the lattices, and the arrows the generator vectors.



$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
$\begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$
$\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$
$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$
$\begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}$
(a)
(b)
(c)
(d)
(e)

Figure 5·2: Some integer lattices and their generator matrices

STEP uses only integer lattices

In the general theory of point lattices, the generator vector elements need not be integers, rather they can be arbitrary rational or real numbers. However, for a variety of reasons, we restrict STEP lattice allocations to the integers.

As we will see in the next section, this choice makes lattice computations much simpler. In particular, we can represent lattices in a canonical form called the Hermite normal form (HNF). This form significantly simplifies various geometric computations.

Another reason for choosing integer lattices is that integer lattices are the only types of lattices that can be embedded in the grid. In fact, all integer lattices are *sublattices* of the grid. Integer lattices admit numerically precise coordinates that can be represented with absolute precision and do not require floating-point approximations.

Finally, integer lattices can, by appropriate scaling, be made to approximate any lattice with an arbitrary degree of precision. As an example consider a hexagonal lattice with the generator matrix $\begin{bmatrix} \frac{\sqrt{3}}{2} & \frac{1}{2} \\ 0 & 1 \end{bmatrix}$.

Next to the most common "square grid", a hexagonal lattice is probably the second most popular type of two-dimensional lattice in CA models. The hexagonal lattice has the advantage of having more symmetries than the square grid, and it also represents a dense spherical packing of points in two dimensions; models such as the FHP lattice-gas (U. Frisch and Pomeau, 1986) leverage this.

No uniform scaling of the hexagonal lattice's generator matrix can convert it to an integer matrix. In such cases, one must find an an approximate generator matrix for an integer lattice. Although the exact geometry is lost, the matrix $\begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$ is a reasonable approximation because it maintains the same group relationships among the generators. (If we call x the X direction generator and y the Y direction generator, one can determine that the relation $\mathsf{x} - \mathsf{y} = \mathsf{y}$ holds for both the original hexagonal lattice and its integer approximation.) Arbitrarily close approximations can be found, but require further scaling of the integer lattice.

HNF generator matrices: a convenient canonical representation

In Fig. 5·2 each lattice has a corresponding generator matrix. Notice that all of the generator matrices have a common form. They are are all upper triangular and have positive elements. While it is possible to represent the lattices with other generator matrices, these lattices are all in a canonical representation, the Hermite normal form (HNF), that STEP employs to represent lattices[4].

Consider the generator matrices of Fig. 5·3. All of them specify the same lattice—the one in Fig. 5·2 (b). Although they all represent the same lattice, only the one on the left is in the HNF that STEP employs. The others are invalid specifications in STEP.

$$\underset{\text{valid}}{\begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}} \quad \underset{\text{invalid}}{\begin{bmatrix} -1 & 1 \\ 1 & 3 \end{bmatrix}} \quad \underset{\text{invalid}}{\begin{bmatrix} 2 & 0 \\ 1 & 1 \end{bmatrix}} \quad \underset{\text{invalid}}{\begin{bmatrix} 1 & -3 \\ 0 & 2 \end{bmatrix}} \quad \underset{\text{invalid}}{\begin{bmatrix} 1 & 3 \\ 0 & 2 \end{bmatrix}} \quad \underset{\text{invalid}}{\begin{bmatrix} -1 & 1 \\ 0 & 2 \end{bmatrix}}$$

Figure 5·3: Valid versus invalid generator matrices

Because the generator vectors for a given lattice are not unique, there are many equivalent generator matrices for a given lattice. Every lattice has a whole set of generator matrices that, mathematically speaking, are sufficient for identifying it.

Practically, however, there is no reason to permit all possible descriptions of a lattice, and, in fact, it's preferable to have a unique, immediately identifiable description. Fortunately, there is a restricted generator matrix form that uniquely identifies an integer lattice. And this form has the added benefit of making a number of computations that STEP needs to do much simpler. This form is called a Hermite normal form (HNF). Intuitively, this form requires an upper-triangular generator matrix with strictly positive elements. This formulation has a mathematical basis

[4]It is worth noting that the HNF is used in other parallel computing related contexts (Wolfe, 1989).

which we now describe.

Paraphrasing from (Weisstein, 2006a) and, given an arbitrary generator matrix A a square $n \times n$ nonsingular integer matrix, there exists an $n \times n$ unimodular matrix U and an $n \times n$ matrix H (known as the Hermite normal form of A) such that

$$\mathsf{AU} = \mathsf{H}$$

As (Weisstein, 2006b) states, "a necessary and sufficient condition that a linear transformation transform a lattice to itself is that the transformation be unimodular." Therefore, because U is a unimodular transformation matrix, any generator matrix can be mapped to an HNF matrix and any lattice lattice may be represented in HNF.

Now, specifying a certain set of conditions on the elements of H makes it unique. Our requirements are that

- $h_{ij} = 0$ for $j > i$
- $0 < h_{ii}$ for all i, and
- $0 \leq h_{ij} < h_{ii}$ for $j < i$.

Basically, this means that the generator matrix

- be upper-triangular, with generators stacked from the most-significant dimension to the least,
- have strictly positive diagonal 'size' elements in each dimension,
- and have non-negative, skew elements that are smaller than the size of the generator in the dimension they skew.

Geometrically, these constraints amount to requiring that the generator matrix use short, positive, axis-aligned generator vectors stacked in order from the highest dimension to the lowest. Figure Fig. 5·3 demonstrates the differences between a valid generator matrix and some invalid ones.

As an example of a unimodular transformation from an invalid, non-HNF matrix to a valid matrix, consider the non-HNF generator matrix $\begin{bmatrix} 2 & 4 \\ 0 & -3 \end{bmatrix}$ this may be converted to a HNF as $\begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix}$ by a unimodular transformation that 1) negates the X generator vector to make it positive and 2) adds the original X generator to the Y generator in order to obtain an X-dimension skew in the Y generator that's less than the X dimension generator.

Although STEP and the SIMP programming environment could use an algorithm to convert arbitrary matrices to HNF, there's really no need for that—the HNF we've selected is a natural, intuitive way to specify a lattice. Furthermore, an automated conversion technique would introduce redundancy to the paradigm because it would allow the same lattice to be represented in multiple ways.

5.1.3 Representing cosets with the rectangular unit cell

Recall from Chapter 4.5.1 that STEP employs a lattice's *rectangular unit cell* as a basis for selecting sites and representing the cosets of a lattice. The rectangular unit cell is a rectangular region that, when translated to the sites of the lattice, covers the n-dimensional space without overlapping.

The rectangular unit cell is obtained from the projection of the generator vectors, expressed in HNF, onto an orthogonal basis. In particular, the rectangular unit cell is given by the diagonal of the HNF generator matrix G. The diagonal vector, g, of the generator matrix G is

$$\mathsf{g} = (g_{00}, g_{11}, \ldots, g_{n-1})$$

The rectangular unit cell, $\mathcal{U}(\Lambda)$, of a lattice Λ with a generator matrix G and a diagonal g is the region

$$\mathcal{U}(\Lambda) = \{\mathsf{v} : 0 \leq \mathsf{v} < \mathsf{g}\}$$

Because this region is a unit cell of Λ it's guaranteed to contain one and only one lattice site.

STEP represents the coset that a lattice allocation (such as a `Signal` or `Event`) inhabits by the position of site within the rectangular unit cell $\mathcal{U}(\Lambda)$. If the lattice has been shifted from the origin by a vector v, the coset representative c is the point of the shifted lattice that intersects with the rectangular unit cell, or

$$\mathsf{c} = (\Lambda + \mathsf{v}) \cap \mathcal{U}(\Lambda)$$

where $\cap$ is the intersection operator. Note that the integer vectors in $\mathcal{U}(\Lambda)$ give the possible coset representative coordinates. The size of $\mathcal{U}(\Lambda)$ (the product of the elements of g) gives the number of possible coset locations.

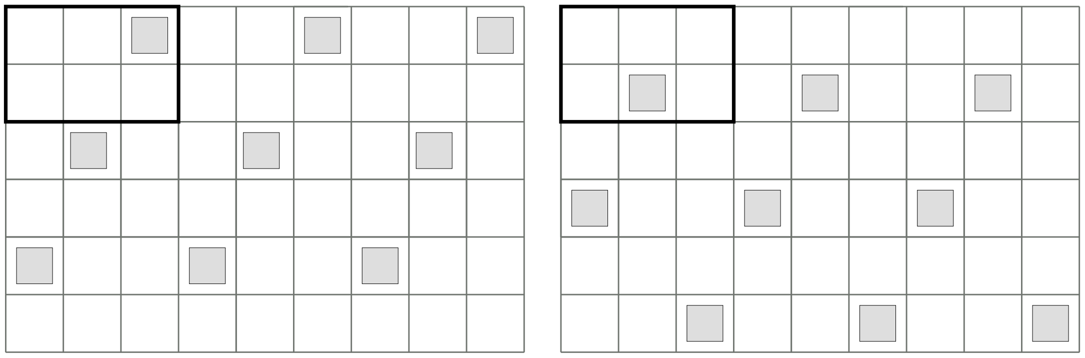


Figure 5·4: Two cosets of a lattice

For concreteness, consider Fig. 5·4. It depicts two cosets of a lattice with a generator of $\mathsf{G} = \begin{bmatrix} 2 & 2 \\ 0 & 3 \end{bmatrix}$. The dark rectangle is the rectangular unit cell as given by the diagonal of the generator, $\mathsf{g} = (2, 3)$. The first coset $(0, 2)$ represents many different shifts of the lattice including $(0, 2)$, $(-2, 0)$ or $(-1, 0)$. The second coset, $(0, 2)$, might have been reached by shifting the allocation by, say, $(-1, 1)$ or, perhaps, $(1, 1)$. (For any given coset, there are a number of shifts that could and the allocation on any other given coset.)

5.1.4 Accessing data

When we talk about accessing data there are two levels—the logical level of the coordinates and the physical level of the indexes in memory. Although the user will not ever directly access the data as it's stored in memory, it will be necessary to transfer data stored in a `Signal` to and from programmer-accessible multi-dimensional array objects. When state variables have an orthonormal generator vector, the mapping from coordinates to array indexes is direct. When it's not, the mapping is less direct.

Single coordinate selection

As discussed in Chapter 4.5.2, an individual coordinate may or may not hit a lattice site. Therefore, to ensure that a coordinate always selects a lattice site, all coordinates select regions that are multiples of the rectangular unit cell.

When the user selects a single coordinate such as `c[2,3]`, STEP implicitly interprets coordinate v as selecting *the* lattice site inside the region $\mathsf{v} + \mathcal{U}(\Lambda)$ where

Region selection

In STEP, regions are expressed by two vectors (u, v) that specify a region

$$\mathcal{R}(\mathsf{u}, \mathsf{v}) = \{\mathsf{w} : \mathsf{v} \leq \mathsf{w} < \mathsf{v}\}$$

All regions must be multiples of the rectangular unit cell. The size array for the region is

$$\mathsf{s} = (\mathsf{u} - \mathsf{v})/\mathsf{g}$$

where division is carried out element-wise. If the size is an integer vector, that is, it has all integer elements, then the region is a multiple of the rectangular unit cell.

The array has data at indexes q such that $0 \leq \mathsf{q} < \mathsf{s}$. The lattice allocation has sites at coordinates $\mathsf{x} \in \Lambda + \mathsf{c}$ where c is the allocation's current coset. The index q of a site at coordinate x is

$$\mathsf{q} = \lfloor (\mathsf{x} - \mathsf{v})/\mathsf{g} \rfloor$$

5.1.5 Implications for events

Not all `Event` and `Signal` lattice combinations are compatible with the basic requirement that events be locally uniform. Consider what would happen if one were to define an event that reads a signal that is in a sublattice of the event—for concreteness consider an event whose lattice is the grid that reads a signal whose lattice is a proper sublattice of the grid. The sites that the event would end up reading would not be at uniform offsets from the event. Rather, multiple event sites would end up reading the same signal site.

In order for an event to act uniformly on the state variables, *the event's lattice must be a sublattice of the state variables it accesses.* If this did not hold, then the action of the events would not be uniform.

Fig. 5·2 shows five sublattices of the 2D grid and their corresponding generator matrices. For each lattice represented in Fig. 5·2, Table 5.1 has a row and a column. The table marks with an 'x' when a row is a sublattice of the column—(a) is a sublattice only of itself; (b) is a sublattice of itself and (a); (d) is a sublattice of itself, (c),(d), and (a); (e) is a sublattice of itself, (c), and (a).

Mathematically, a lattice Λ is a sublattice of another lattice Γ if all of the sites of Λ are also in Γ. In terms of the generators, a Λ is sublattice of Γ if the generator

	a	b	c	d	e
a	x				
b	x	x			
c	x		x		
d	x	x	x	x	
e	x		x		x

Table 5.1: Table giving the sublattice relations of the lattices in Fig. 5·2

vectors of Λ can be expressed as integer combinations of the generator vectors of the generators of Γ.

If the generators are in HNF, then the following is a necessary and sufficient condition for the lattice generated by A to be a sublattice of B

$$\mathsf{A}\mathsf{B}^{-1} \in \mathbb{Z}^n$$

5.1.6 Write conflicts

Not every set of output signals for an event is valid. In particular, a given output can only be written once, otherwise we have a *write conflict.* A write conflict occurs if an event tries to write the same location of a signal twice.

As an example, consider the following SIMP event transition function

```
def foo():
   a[0]._ = 1
   a[1]._ = 1
```

The output set is $\{(\mathsf{a},[0]),(\mathsf{b},[1])\}$. If the lattice of both the event and the signal is the grid, then the output values of neighboring local events would collide. Because the local functions are all applied in parallel, there is no preferred way to handle this situation, and the output is invalid. If, however, the lattice of the event were, say, a sublattice with the generator $\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$, then the output set would be perfectly valid.

So, a write conflict may exist if an event writes the same signal more than once—that is, the same signal appears in the output set multiple times. It's the STEP API caller's responsibility to detect this and prevent write conflicts. If there is a write conflict, the STEP is free to raise an error or to just resolve the conflict using any strategy it deems appropriate.

Considering all possible coset positions for the signal and the event, a write conflict may happen if the unit cell regions selected by the different outputs overlap. Otherwise, it's not possible. In particular, given an event lattice Λ, a signal lattice Γ, and a set of output offset vectors $\mathsf{v}_0, \mathsf{v}_1, \ldots$ of the signal we would like to know whether or not the sets $\mathsf{v}_0 + \Lambda + \mathcal{U}(\Gamma), \mathsf{v}_0 + \Lambda + \mathcal{U}(\Gamma), \ldots$ intersect or are disjoint. If they interset, there is a potential for a write conflict, if they are disjoint, there is no such potential.

A simple way to determine wheter there is an overlap is to compute the points $\mathsf{v}_i + \Lambda + \mathcal{U}(\Gamma)$ for each output offset vector and determine whether any of them are the same. Fortunately, it's not necessary to do this for each point in the lattice, rather, its sufficient to just map the points in the region $\mathsf{v}_i + \mathcal{U}(\Gamma)$ to their representative coset coordinates in Λ. In practice, this can be done by constructing what we call an *output coset table*. The output coset table has an entry for each coset coordinate k in the event's unit cell $\mathcal{U}(\Lambda)$. In order to detect overlaps, one can just loop over v_i, inserting an indicator for each coordinate w_i in $\mathsf{v}_i + \mathcal{U}(\Gamma)$ into the table, first checking to see whether there is already an indicator in the table. If it's there, then an error giving the two offsets that might conflict can be raised.

5.1.7 Wrap-around compatibility

All lattices for signals and events must be compatible with the size of the torus that was declared at initialization time. In particular, as mentioned in Chapter 4.5.3, a lattice must wrap around to itself rather than wrapping around to a different coset.

A necessary an sufficient condition for this is that the size vector d actually contain a lattice site. This will be true for a lattice with a generator G if and only if $\mathsf{d}\mathsf{G}^{-1}$ is an integer vector.

A rectangular boundary must have a lattice point, otherwise, the lattice does not wrap-around. One may view the bounding vectors of the torus as generators of a "bounding lattice". In this view, the coordinates on the torus wrap-around as cosets of the bounding lattice.[5] In order for lattices allocated on the torus to wrap-around to themselves, they must have the bounding lattice as a sublattice.

Whenever one tries to declare a signal or event on an incompatible sublattice, SIMP raises an error and suggests an alternate choice of boundary size. We now discuss how SIMP does this.

In order to determine if a lattice is wrap-around compatible, SIMP just divide the (rectangular) size vector of the space by a vector that gives the the bounds of the least common rectangular unit cell. If the result is not an integer, then we suggest a size rounded up to the next integer lattice.

Recall from Chapter 4.5.3 that the least common rectangular unit cell is the smallest rectangular unit cell on which the lattice repeats under a rectangular tiling. The least common orthogonal (rectangular) lattice can be computed as

$$q_{ii} = g_{ii}\mathrm{lcm}(\{g_{jj}/\mathrm{gcd}(g_{jj}, g_{ij}) : g_{ij} \neq 0\})$$

[5]Even if the boundaries were not periodic, this requirement is necessary in order to ensure that there are the same number of lattice sites in each dimension regardless of what coset the lattice has been shifted.

where lcm computes the least common multiple and gcd computes the greatest common divisor. The computation can be understood as follows. The term $g_{jj}/\gcd(g_{jj}, g_{ij})$ computes the minimum number of repetitions of generator g_i needed before g_i returns to a location that's equivalent to the location of the orthogonal projection of the g_j generator. To compute q_{ii} we take the least common multiple of this number for all directions j and multiply the rectangular component g_{ii} by this number.

5.2 The STEP computational model

Having defined the geometry here, and in previous chapters given background on the types of computation STEP is meant to support, we are now in a position to define the abstract STEP processor model. That's what this section does.

The STEP processor consists of a global topology, state variables allocations, operations such events, shifts, shuffles, reads and writes.

Topology

A STEP has a global topology defined by a discrete, n-dimensional, orthonormal integer lattice that we call the *grid*. The sites of the grid are the integer vectors x in $\mathbb{Z}^n$. Coordinates, x, are are given as vectors in (z, y, x) form—with the most significant dimension first. The global topology is finite, bounded by a positive vector d. Coordinates are taken modulo d (x_i equivalent to $x_i + kd_i$ for any integer k). The global topology is declared at initialization time by declaring d.

Signals

After initialization, one can allocate state variables called *signals* on the grid. Signals are parallel state-variable allocations with a uniform type allocated at every grid site or at regular intervals defined by *sublattices* of the grid. A sublattice Λ of the grid is represented by a positive, non-singular, upper-triangular, integer *generator matrix* G, as described in Section 5.1.2, and can be expressed as

$$\Lambda = \{\mathsf{aG} : \mathsf{a} \in \mathbb{Z}^n\}.$$

In all, a signal is defined by its type, its lattice, and the current coset that it inhabits. (A *shift* operation can move a signal from one coset to another.) The following completely describes a signal

$$s = \{\Lambda, \mathcal{T}, \mathsf{c}\}.$$

A signal s has values of type $\mathcal{T}$ at all points k in $\Lambda + \mathsf{c}$. That is,

$$s[\mathsf{k}] \in \mathcal{T} : \mathsf{k} \in \Lambda + \mathsf{c}.$$

The sites k of a signal are a subset of grid coordinates x. A grid coordinate does not always directly ‘hit’ a site. Nevertheless, STEP defines a means by which each coordinates x maps to a coordinate k of a signal s. In particular, the site selected by $s[\mathsf{x}]$ is equivalent to $s[\mathsf{k}]$ where k is the site that intersects with the the region given by the *rectangular unit cell* of Λ translated to x. The rectangular unit cell is a region that’s guaranteed to contain one and only one site in Λ. The rectangular unit cell $\mathcal{U}(\Lambda)$ of a lattice Λ with a HNF generator G with a diagonal g is defined as

$$\mathcal{U}(\Lambda) = \{\mathsf{r} : 0 \leq \mathsf{r} < \mathsf{g}\}.$$

The overall expression for selecting a site at a generic coordinate x is

$$s[\mathsf{x}] \equiv s[\mathsf{k}], \mathsf{k} = (\Lambda + \mathsf{c}) \cap (\mathsf{x} + \mathcal{U}(\Lambda)).$$

Events

An *event* is a uniform, parallel, CA-like update. Like a signal, it can be defined on the grid or on a sublattice of the grid. It may also shift from one coset to another.

Similar to a signal, the global structure of an event is represented by a lattice and a coset.

Issuing an event uniformly applies it at all sites. Locally, an event reads a set of inputs, uses the input values to compute output values, and writes the outputs. The local structure of an event is represented by a set of inputs and output and a function that maps input values to output values. The inputs and outputs are represented by pairs signals and their local offsets.

In particular, an event E consists of a lattice Λ, a coset c, a function, f, and inputs 'in' and outputs 'out'

$$E = \{\Lambda, \mathsf{c}, f, \mathrm{in}, \mathrm{out}\}.$$

The 'in' and 'out' sets are defined by n-input signal/offset pairs and m-output signal/offset pairs. In particular

$$\mathrm{in} = \{(s^{<\mathrm{in}_0>}, \mathsf{v}^{<\mathrm{in}_0>}), (s^{<\mathrm{in}_1>}, \mathsf{v}^{<\mathrm{in}_1>}), \ldots, (s^{<\mathrm{in}_{n-1}>}, \mathsf{v}^{<\mathrm{in}_{n-1}>}); \}$$

$$\mathrm{out} = \{(s^{<\mathrm{out}_0>}, \mathsf{v}^{<\mathrm{out}_0>}), (s^{<\mathrm{out}_1>}, \mathsf{v}^{<\mathrm{out}_1>}), \ldots, (s^{<\mathrm{out}_{m-1}>}, \mathsf{v}^{<\mathrm{out}_{m-1}>}).\}$$

The transition function f is an n-input, m-output function that maps input values to output values. The function has the following type signature:

$$\begin{aligned} f: \quad & \{\mathcal{T}(s^{<\mathrm{in}_0>}), \mathcal{T}(s^{<\mathrm{in}_1>}), \ldots, \mathcal{T}(s^{<\mathrm{in}_{n-1}>})\} \longrightarrow \\ & \{\mathcal{T}(s^{<\mathrm{out}_0>}), \mathcal{T}(s^{<\mathrm{out}_1>}), \ldots, \mathcal{T}(s^{<\mathrm{out}_{m-1}>})\}. \end{aligned}$$

Overall, an event E may be written as follows

$$\mathrm{out}[\mathsf{x}] \leftarrow f(\mathrm{in}[\mathsf{x}]), \mathsf{x} \in \Lambda + \mathsf{c},$$

where

$$\mathrm{in}[\mathsf{x}] = \{s^{<\mathrm{in}_0>}[\mathsf{x}+\mathsf{v}^{<\mathrm{in}_0>}], s^{<\mathrm{in}_1>}[\mathsf{x}+\mathsf{v}^{<\mathrm{in}_1>}], \ldots, s^{<\mathrm{in}_{n-1}>}[\mathsf{x}+\mathsf{v}^{<\mathrm{in}_{n-1}>}]\}$$

and

$$\mathrm{out}[\mathsf{x}] = \{s^{<\mathrm{out}_0>}[\mathsf{x}+\mathsf{v}^{<\mathrm{out}_0>}], s^{<\mathrm{out}_1>}[\mathsf{x}+\mathsf{v}^{<\mathrm{out}_1>}], \ldots, s^{<\mathrm{out}_{m-1}>}[\mathsf{x}+\mathsf{v}^{<\mathrm{out}_{m-1}>}]\}.$$

.

For uniformity's sake, the lattice Λ of an event is required to be a sublattice of all lattices Γ of the signals in the 'in' and 'out' sets. This constraint holds if-and-only-if AB^{-1} is an integer matrix for the generator matrix B of the event for all generator matrices A of the signals it accesses.

Shifts

Both signals and events can be shifted. A shift is defined by both the signal or event to be shifted and the vector that gives the shift amount. Shifting a signal moves its data and updates the coset it inhabits. Shifting an event just updates its coset.

Because the effect on the coset is common, let's cover it first. When a shift of v is applied, it shifts the lattice of a signal or event from $\Lambda + \mathsf{c}$ to $\Lambda + \mathsf{c} + \mathsf{v}$. Because the coset representative is the lattice site in the lattice's rectangular unit cell, the new coset representative c' is[6]

$$\mathsf{c}' \leftarrow (\mathsf{c} + \mathsf{v} + \Lambda) \cap (U)(\Lambda)$$

[6] The expression we use to perform this computation is the following

$$\mathsf{c}' \leftarrow \mathrm{mod}(\lfloor(\mathsf{c}+\mathsf{v})/\mathsf{g}\rfloor \mathsf{G} + \mathrm{mod}(\mathsf{c}+\mathsf{v}, \mathsf{g}), \mathsf{g}),$$

where G is the generator matrix for the lattice and g is its diagonal and the division and modulus operators are carried out element-wise.

Shifting a signal both updates its coset and actually moves data. The effect of shifting a signal s by an amount v is

$$s'[\mathsf{k}+\mathsf{v}] \leftarrow s[\mathsf{k}], \mathsf{k} \in \Lambda + \mathsf{c}$$

where indexing the updated signal $s'[\mathsf{x}]$ uses the updated coset c'.

I/O operations

The basic read and write operations are relatively simple operations that map rectangular signal regions to dense arrays with integer indexes. In particular, we are interested in mapping region $\mathcal{R}(\mathsf{u},\mathsf{v})$ of a signal s to a rectangular array A. The region $\mathcal{R}(\mathsf{u},\mathsf{v})$ is defined as

$$\mathcal{R}(\mathsf{u},\mathsf{v}) = \{\mathsf{r} : \mathsf{u} \leq \mathsf{r} < \mathsf{v}\}.$$

The size of the region is $\mathsf{v}-\mathsf{u}$. It maps to an array of size $\mathsf{z} = (\mathsf{v}-\mathsf{u})/\mathsf{g}$ where g is the size of the signal's lattice's rectangular unit cell. (Because the size of the array must be an integer, a region's size must be a multiple of the rectangular unit cell, that is, $(\mathsf{v}-\mathsf{u})$ must be a multiple of g.)

The mapping from the signal to the array is as follows

$$A[\mathsf{i}] = s[\mathsf{u}+\mathsf{i}*\mathsf{g}] : 0 \leq \mathsf{i} < \mathsf{z}.$$

In addition to the normal signals that actually require a STEP to allocate and maintain state there are also special I/O signals that don't require storage. They just provide entry and exit points for reading and writing values through events. In our current implementation we provide a combined read event that can be used to perform rendered reads. Rendered reads I/O signals following an event that renders state into the signals. I/O signals have all the same properties as ordinary signals, except that the STEP does not store their state. They can be shifted and used in events; however, it is only valid to read an I/O signal in an event if it immediately follows a write and it is only valid to read an I/O signal if the read it immediately follows an event.

Stirring

Another operations is the *stir* operation. Stirring is like a non-local, random shift, but with the exception that it need not actually be a shift, but may be some other non-local permutation. Because, in a given memory layout, one permutation may be cheaper than another, an implementation has the freedom to choose which permutation to employ. The only constraint is that it should attempt to make sure that a sequence of stirs is also temporally non-local—the same data should not arrive in the same place too soon or at regular intervals.

5.3 The STEP API

In this section, we provide a description of the STEP API that highlights interesting aspects, but stop short of documenting all details of the programming interface. For that, see the SIMP Reference Manual (Bach, 2005).

The STEP API is an abstract interface to an abstract hardware/software coprocessor. It is not a general purpose programming language. The reason is simple—a CA processor is not a good model of general-purpose computing; for that one should employ von Neumann's more famous architecture, the standard load/store architecture.

Let's discuss the STEP operations. First an operation is declared and then it may be called. The instantiation is a first "compilation" step, at which point errors may be raised if there is a problem with the declaration of the operation. The instantiation is handled both by the abstract STEP classes and by the STEP implementation, which has a chance to do some pre-processing and compilation at declaration time.

The idea is that, because most operations will typically be called over and over again, a STEP implementation should take the opportunity to do optimizations. However, if the optimizations are expensive, it is also possible for a STEP implementation to cache previously compiled code and optimizations. For example, the transition function can be easily cached.

The STEP architecture is *permissive* in that allows a STEP to implement a subset of the operations and throw an exception when a CA application tries to declare an operation or state variable type that it can't implement. This is important because certain hardware implementations may be resource-limited and not able to handle certain sizes or numbers of dimensions. For example, our pure lookup-table based implementations can only have small, bit-sized types as state variables. (Note that the CAM-8 was restricted to a default maximum of 16 total bits of state, required that transition functions be specified in 16 or fewer bits of state, and restricted the length of each dimension to a power of 2.)

Because a STEP requires the freedom to store state variables in any way that it finds fit, or may even store the state variables off machine, the actual storage of state variables is entirely hidden. The API encapsulates the storage details of signals. State variables must be accessed through special read and write API operations.

The STEP API manages the allocation of state variables and defines a number of types for state variables. These types include binary unsigned integers with 1 through 7 bits; 8-, 16-, and 32-bit signed or unsigned integers; and 32-bit floating-point numbers.

The exact state and memory are an implementation detail that the user doesn't need to see. We hide the state and implementation because the layout of the data may be complicated. Forcing one format might unnecessarily restrict memory layout optimizations such as bit-packing and data partitioning for cache and CPU locality. The state may not even be local to the PC. It may be on other another machine altogether (e.g. multi-computer, or external hardware implementation) or only accessible through a driver interface (e.g. CA implemented on a FPGA board). Even if the state is local and stored in some common, documented format, one would need a mechanism for synchronizing access. The STEP API provides for all of this.

5.3.1 The API's implementation

Unlike OpenGl which is a light-weight API based upon C, we based our implementation of the STEP API on the Python programming language, which allowed us to develop it rapidly and provide a flexible interface to our STEP implementations which, also were primarily coded also in Python as well.

Originally, we had started with a pure C implementation of the STEP API that was very closely based upon the CAM-8's STEP interface. Transition functions were represented as lookup tables, and all operations were required to be expressed in terms of pure lattice gas operations—as shifts that translate signals and events that operate on them.

We ended up abandoning this approach when we realized that this interface, although efficient, was quite cumbersome for the programmer and would require a massive overhaul in order to support extensions such as non-orthogonal geometries. We made the API more generic and re-cast our software to include non-orthogonal lattice constructs.

Later, recasting the STEP API in C might be a reasonable thing to do if the interface were to become standardized enough. Recasting the API in C could make it more light weight and easier to bind to other languages. Challenges include picking a generic array format, creating the attendant data structures for defining shifts, and defining a generic language for specifying transition functions.

5.3.2 Class interface

The following defines the STEP class interface.

class `Step(size,args)`

The base constructor does not have any default keyword arguments, but, an individual implementation may define some. The constructor should have default values for any arguments that are not specified explicitly.

`size`

vector giving the size of the grid—the coordinate space on which allocations are made.

`args`

Optional Python dictionary that supplies arguments to the STEP. This can be used to pass implementation-specific arguments to the STEP. One example is the `maxlutsize` argument to the PcCodeGen step, another is the `verbose` argument which makes the STEP report extra debugging information, a final example is the `clearcache` argument which, for debugging purposes, clears any objects (LUTs, etc) cached on disk.

Primary Methods

`declare(step_object)`

Declare state variable (`Signal`) or a user-defined operation. When an object is declared the STEP checks to see whether it can implement the operation or allocate the requested `Signal`. If not, it raises a `StepError` indicating the problem.

`issue(op)`

Issue an operation. User-defined operations must be declared before they're issued.

5.3.3 STEP operations and constructs

Most of semantic content is in the arguments to the `declare` and `issue` methods. The arguments are Python classes that the API defines. The classes have a set of member data elements that define the operation. To use the classes, the API client first creates a class instance and fills in appropriate values for the member data, then the client declares the class to the STEP by passing it as an argument to the `declare` method.

Table 5.2 briefly enumerates the STEP API objects and documents their member data. The meanings of these operations were given in the previous section.

`Event`	$(f : \text{in} \mapsto \text{out})$	
	`generator`	G, generator matrix
	`input`	in, list of n input signal/offset tuples
	`output`	out, list of m output signal/offset tuples
	`function`	f, function with n input arguments that returns m outputs.
`Shift`	*shift signals and events*	
	`shifts`	list of pairs of (signal or event) objects to shift and shift vectors
`Stir`	*non-local permutation on data, randomize coset*	
	`objects`	list of Signal/Event objects to stir

Table 5.2: STEP Dynamics Operations

Table 5.3 lists the signal, I/O operation, and control operation classes.

The signal classes can not be issued, but they can be declared. As we discuss shortly, the `IoSignal` class is for reading and writing values into and out of events. It's a useful mechanism for passing data into and out of events without actually having to declare a signal that the STEP will persist.

The I/O Operation classes include the `Read` and `Write` operations as well as the `SetCoset` and `GetCoset` operations. The first two are for transferring data into and out of signals while the second two are for accessing the coset positions of signals and events.

The control operations provide synchronization methods and allow one to create batch sequences of events. There's an operation called `Flush` that makes the STEP block until all previously issued operations have been committed. One can also submit a `Callback` operation that the STEP will call after all preceeding I/O operations have completed and before subsequent I/O operations start. The `Callback` operation is useful for modifying I/O buffers in the middle of a sequence of STEP operations.

There is also a control operation called `SeedRandom` for setting the random seed used for the `Stir` operation and any future randomized operations. The default seed is `0`. One can use this for generating repeatable stochastic experiments. A STEP is required to perform the same shuffles given a certain seed.

5.3.4 Signal types

A signal's type is declared with another set of STEP classes. For small integer types, one can use the parameterized `SmallUInt` type. It is declared with the number of possible values that it can take on. (This is preferable to the number of bits because it potentially constrains the size of the lookup table.)

class `SmallUInt(n=2)`

> The STEP parameterized small unsigned integer type. It can take on n possible values from 0 to `n-1`. The value of `n` should be less than or equal to 256. When `n=256` `UInt8` is a better choice.

The other types come from `numarray`. They are

Signal Classes (declared but not issued)

`Signal`	*Declare a signal (can not be issued)*	
	`generator`	generator matrix
	`type`	
`IoSignal`	*Declare a stateless* ***`Signal`*** *for I/O operations*	

I/O Operations

`Read`	*Read signals into array buffers*	
	`signal`	list of signals to read
	`region`	region to read
	`buffer`	list of array objects to read state int
`Write`	*Write signals from array buffers*	
	`signal`	list of signal to read
	`region`	region to read
	`buffer`	list of array objects containing state to write
`GetCoset`	*Read current coset of a signal or event*	
	`objects`	list of signal/event objects cosets to read
	`buffer`	
`SetCoset`	*Set the coset of a signal or event*	
	`objects`	list of signal/event object cosets to write
	`buffer`	

Control Operations

`Sequence`	*Perform a sequence of STEP operations*	
	`operations`	sequence of operations to perform
	`repeat`	number of times to repeat the sequence
`Flush`	*Block until pending STEP operations complete*	
`Callback`	*Called after preceeding I/O ops complete, before subsequent begin*	
	`function`	the callback function
	`args`	arguments to be passed to the callback function
`SeedRandom`	*Seed the random number generator*	
	`seed`	integer containing the seed to use

Table 5.3: STEP signal and I/O operation and control operation classes

`UInt8` 8-bit unsigned integer

`Int8` 8-bit signed integer

`UInt16` 16-bit unsigned integer

`UInt16` 16-bit signed integer

`UInt32` 32-bit unsigned integer

`Int32` 32-bit signed integer

`Float32` 32-bit IEEE floating point number

5.3.5 Control and I/O synchronization

Because a STEP acts as a co-processor and does not make control decisions it does not know the exact structure of the applications-level control and decision points or make direct optimizations such as branch prediction with respect to them. Similar to OpenGl, we mitigate this apparent deficiency by introducing a SEQUENCE operation for declaring and issuing batch sequences of STEP operations. An example of where this kind of buffering is useful is in implementations that take a LGA-like approach to implementing cellular automata and must prepare data to be sent to neighboring sites in the next transition function as part of the current transition function.

Despite the fact that a STEP can not know ahead-of-time everything that it should do as a batch process, the STEP API provides a means for the programmer to batch sequences of operations and provides an asynchronous issue semantics. Issuing a STEP operation does not normally cause the issuer to block. Rather, the `issue` method may return immediately, regardless of whether the STEP has yet completed the operation that was issued. In this way the controlling program can continue executing and issuing further operations while the STEP is busy running. A STEP engine can employ the asynchronous execution semantics to buffer up operations, combining them where appropriate. Applications can employ the asynchronous semantics to prevent the STEP co-processor from idling while the application is figuring out what operation to issue next. Applications can also employ the semantics to implement the observer/subscriber pattern with the STEP issues a `Callback` notification operation when it has data ready to be displayed.

The buffer classes have a simple interface. In order to synchronize access between the STEP and SIMP, the buffer class protects the buffer contents with a lock. Before reading data into or out of the buffer, the API client is required to lock the buffer. The buffer lock duration should be as minimal as possible. At declaration time, the the buffer's type (`buffer_type` attribute) must be set. The type options are constant, static, or dynamic. Constant means that the data in the buffer does not change. This is only valid for the `Write` and `SetCoset` operations which read the buffer values. Static means that the buffer's will always be the same memory locations—the client will not change the identity of the buffer. Finally, a dynamic buffer can be changed to alternate memory locations.

One mechanism that STEP provides is the `Flush` operation. When issued, this operation actually does block until all operations that precede it complete. After a `Flush`, one can be sure that any `Read` operations issued before the `Flush` have completed. This level of control is sufficient, but is not very fine-grained.

For finer control, one may also issue *callback* functions as STEP operations. In order to implement `Callback` operations, the API allows one to *issue* Python functions or include them in issued sequences. The STEP API requires the STEP to call any functions issued *after* all preceding I/O operations have completed, and *before* any subsequent I/O operations begin. As such, a callback function is a handy way to 1) perform an action after data has been read, 2) prepare data in a write data buffer just before it is written or 3) modify the identity of a buffer just before an I/O operation is issued.

IoSignals and rendering

A special kind of signal, called an `IoSignal` can be used for rendering purposes.

One would often like to apply a rule to do a rendering operation. If one used regular signals to hold the output values and then read the values out of these signals, extra storage would be needed—internal storage for the signals and an external array for the output data. In addition, an extra copy would be needed—a copy from the signal to the output array. Two copies must be kept, because, it is assumed that signals will be used again as inputs to future rules. If one instead declares a special `IoSignal`, a STEP can realize that it the output signal will only be read or written, but does not have persistent state.

IoSignals can be used to construct a combination of operations that performs rendering. In particular, to read rendered values, one constructs a sequence of operations—an `Event` that outputs to `IoSignal` objects, followed by a `Read` that reads those `IoSignal` objects. The STEP can then combine the operations into a single render and read operation, writing directly into the output buffer. It's only valid to read an `IoSignal` immediately after an event that writes it. By the same token, it's only valid to employ an event to read an IoSignal if the preceeding operation was a write.

Chapter 6

SIMP Implementation

While the raw STEP API provides an effective means of controlling a STEP, it is not meant to be a user-friendly programming environment. Writing applications directly with the API would be more labor intensive than writing the SIMP programs demonstrated in Chapter 4.

SIMP makes STEP operations easier to use in the following ways

- SIMP subclasses the STEP classes to make them more user friendly. It provides class constructors that accept the information for constructing the relevant STEP members as arguments. At the end of the constructor, it automatically calls the STEP `Declare` method. It also creates a wrapper for the "Do" call, by making the objects themselves callable.

- Most significantly, SIMP wraps the `Event` class constructor and provides a means of converting Python functions of an event as a function into the input/output port and transition function format that a STEP expects.

- SIMP wraps the signal class providing an array interface with subscripting. With this interface, one may select rectangular signal regions. One may also employ the interface to implicitly generate read and write operations. The array interface can also be employed to declare named CA neighbors as in '`NW = C[-1,1]`'.

- SIMP has a renderer class that coordinates the user interface and STEP-based rendering.
- Usually one only requires a single STEP instance. Since one doesn't usually want to employ multiple STEP implementations, SIMP provides methods for initializing and importing a global STEP module that's implicitly used in all SIMP contexts.
- SIMP simplifies the selection of a STEP instance. It provides a default 'import list' of STEP's ordered by performance. At initialization time it tries to import the first STEP in the list. If it fails, it tries the next, and so on. SIMP also provides means to override the import list at the system, user and program level.

We don't bother to go into all of the details of everything that SIMP does, rather, we choose to focus on the most significant thing—converting Python functions to STEP Events.

6.1 `Event` transition-functions

An `Event`'s transition-function specifies the local function that's evaluated in parallel at every point of the `Event`'s lattice when the `Event` is called. For the most part, a transition-function has the same syntax and semantics as an ordinary Python function. However, for convenience and to allow a STEP to implement an `Event` efficiently (for example, by transforming it into a lookup table or `C` code), a transition-function has some special behaviors and restrictions not present in ordinary Python code. This section discusses these aspects of transition-functions.

6.1.1 The basics

The transition-function is a local function that computes next-state signal values from their present-state values. It reads present-state values from a signal directly and writes next-state values to a signal using the underscore attribute. Inside the transition-function, subscripts are relative to the site being updated; outside they are relative to the origin.

As an example, consider a two-dimensional program with a `Signal`, `c`. Inside a transition-function, the subscripted signal `c[0,0]` gets the value of `c` at an offset of zero from the site being updated. The next-state value of a signal is written using the underscore attribute, as in `c[0,0]._=1`. For example, in the following 2D parity event, we can write

```
c = Signal(SmallUInt(2))
def parity():
   c[0,0]._ = c[0,0]^c[-1,0]^c[0,1]^c[1,0]^c[0,-1]
parity_event = Event(parity)
```

in which next-state value is the binary exclusive-or of the signal and its nearest neighbors.

Implicit zero offset

Most transition-functions access the signal at an offset of zero. Therefore, as a convenience, when a signal is accessed in a transition-function without as subscript the implicit subscript is zero. So, within a transition-function, `c` is equivalent to `c[0,0]` and `parity` may be rewritten as

```
def parity():
   c._ = c^c[-1,0]^c[0,1]^c[1,0]^c[0,-1]
```

In contrast, outside of a transition-function, `c` refers to the values at all sites and `c.value()` returns the entire array of values in `c`.

The present-state and next-state are independent values

In a transition-function, the present-state of a signal remains unchanged even after the next-state value is assigned. Therefore, even if we write

```
def example():
   c._ = 1
   if c==0: # may still be true
      ....
```

the `if` conditional may still be true since '`c._ = 1`' assigns the next-state, but `c==0` references the present-state.

Signal references are variables

Inside a transition-function, present-state and next-state signal references are variables. This stands in contrast to the ordinary context in which `Signal` and `SignalRegion` objects behave like addresses and must be explicitly dereferenced by calling the `value` method.

For example, in a transition-function like

```
def example():
   ne = c[-1,1]
   ne._ = 1
```

an error would be raised because `ne` (short for Northeast) is a variable that holds the

Northeast *value* of c at an offset of [-1,1] rather than a *reference* to this location. When a neighbor reference is needed, the proper way to get it is outside of the transition-function, as in

```
ne = c[-1,1]
def example():
   ne._ = 1
```

The transition-function should only access output values using global names.

6.1.2 Frozen global context

The *global context* of a function is the set of global values it references. The global context of an Event's transition-function is frozen when the Event is created. All global names are replaced by the values they had at that time. When the transition-function runs it uses these values for the globals and, unlike normal Python functions, does not notice if the value associated with a global name changes.

Freezing the global context makes an Event a static entity. This, in turn, allows a STEP to determine the transition-function's precise set of inputs and outputs, and, in the case of a code-generating STEP such as PcCodeGen, perform type analysis.

Parameterized events

By changing context values at the time an Event is constructed one can create Event objects that, despite sharing a common Python transition-function, have different behaviors. We call such events *parameterized events* because they are parameterized on their context.

As an example, we make two different events, grow2 and grow3 that grow domains of signals in state 1 if the sum of nearby sites in state 1 exceed 2 and 3, respectively.

```
def grow():
  sum = c[0,1] + c[1,0] + c[0,-1] + c[-1,0]
  if sum>threshold:  c._ = 1

threshold = 2; grow2 = Event(grow)

threshold = 3; grow3 = Event(grow)
```

Rather than writing statements to assign `threshold`, one can instead use the `context` parameter as in

```
grow2 = Event(grow,context={"threshold":2})
grow3 = Event(grow,context={"threshold":3})
```

`context` is a dictionary that overrides values for names. The `Event` constructor looks for values in the context dictionary first and then in the global namespace.

One can also parameterize a event on the signals it references. For example, suppose we want to run two copies of the parity event and just view the difference in the time evolution of the events. Assuming that we have already allocated two signals, `a` and `b` to hold the two copies of state, we can write

```
def parity():
   sig._ = sig^sig[-1,0]^sig[0,1]^sig[1,0]^sig[0,-1]
parity_a = Event(parity,context=kwdict(sig=a))
parity_b = Event(parity,context=kwdict(sig=b))
step = Sequence(parity_a,parity_b)
```

to create a `Sequence` that updates both copies of the dynamics independently. To render the difference between the two copies we can use a rendering event like

```
white = IoSignal(UInt8)
def difference(): # output white if a!=b
   if a!=b: white._ = 255
difference = Renderer(difference,white)
```

We can watch the spread of a single point difference by initializing with the following code

```
arr = makedist(a.shape,[1,1])  # get a random array
a[:,:] = arr;  # initialize a with it
if arr[0,0]==1: arr[0:0] = 0  # toggle the first element in the array
else: arr[0,0] = 1
b[:,:] = arr  # initialize b to the altered array
```

The Parity CA is discussed in more detail in Chapter 8.3.

6.1.3 Restrictions on transition-functions

In addition to their special behaviors, there some special restrictions on transition-functions. In particular,

- signal subscripts in a transition-function must be constant,
- transition-functions are stateless,
- sub-function calls can not directly reference signals,
- some Python constructs, such as unhandled exceptions and `yield` statements, are not allowed,
- a event must not write parallel conflicting values

These basic restrictions are common to all of the STEP implementations, and enforced by SIMP. We'll be addressing each in turn. Although these limitations are only limitations of the default implementation, a STEP implementation may impose other limitations. Indeed, Section 7.4 discusses the further limitations of the `PcCodeGen` implementation that are necessary for it to generate `C` code on-the-fly. Other possible restrictions are, say, requiring that variables only be assigned only once so that the transition function can be directly translated to a combinational network (Böhm et al., 2002).

Signal subscripts must be constant

Within a transition-function subscripts must evaluate to constants and cannot be a function of present-state values. This is because, for efficient implementation, the input set of a transition-function must be constant. If the subscripts depended on present-state values, the input set would be dynamic and a STEP would have a more difficult time gathering the input set for the transition-function in a homogeneous, optimized way[1].

In particular, the following transition-function is disallowed

```
def badfunc(): # get next state value from the right if c==1
   c._ = c[0,c]
```

but the transition-function below is okay

```
def goodfunc(): # get next state value from the right if c==1
   if c==1: c._ = c[0,1]
```

[1]The current STEP implementations gather the input set of a transition function, execute the function, and then write the output set. Because these sets are static, the gathering and writing is efficient. It is conceivable that, by performing static analysis of the transition-function, a STEP could determine all the possible values of a state-dependent subscript and add inputs for each to the input set. However, the input set may be much larger than the user expects. Alternately, a STEP could implement events with dynamic input sets by avoiding the construction of a static input set and gathering inputs only as needed. However, performing the added address logic for each neighbor access would make evaluating the transition-function many times more expensive.

Transition-functions are stateless

A transition-function can not carry values from one call to the next. A user might be tempted to use mutable objects in the global context to carry values as in

```
carrier_list = [0]
def badfunc():
   c._ = carrier_list[0]
   carrier_list[0] = c
```

which tries hold onto the current state value of `c` in the first element of `carrier_list` and use that value to assign the next state value of `c` the next time that the transition-function is applied. However, because the machinery that processes the transition function does not hold state[2], a transition-function like `badfunc` will, depending upon the STEP implementation, lead either to unpredictable behavior or an error.

Of course, all sub-functions called by a transition-function must also be stateless.

Sub-function calls must not directly reference signals

All references to signal objects must be contained in the top-level of the transition-function. A event will ignore any signal reads or writes carried out within sub-functions called by the transition-function. For example, the following version of parity would not work properly

```
def xor(): return c[0,0]^c[-1,0]^c[0,1]^c[1,0]^c[0,-1]
def badparity():
    c._ = xor()
```

Because, when a constructing an `Event` object, the STEP only analyzes the code

[2]A more fundamental reason for this is that carrying state across transition function invocations creates a scan-order based dependency. But, since CA events are parallel, there is no logical scan order to the individual updates and, therefore, the semantics of carrying data across a scan is not defined.

of the transition-function itself and not the code of any sub-functions that it may call, it would not be aware that `c[0,0]`, `c[-1,0]`, `c[0,1]`, `c[1,0]` and `c[0,-1]` are needed as inputs.

The following shows an allowable use of a sub-function in which signal values are passed as parameters from the top-level transition-function

```
def xor(a,b,c,d,e): return a^b^c^d^e
def goodparity():
    c._ = xor(c[0,0],c[-1,0],c[0,1],c[1,0],c[0,-1])
```

In the future, this restriction may be removed by adding recursive inlining capabilities to the transition-function analysis routines.

6.1.4 Converting high-level functions to STEP events

An implementation would like to see an event in terms of a function and an input/output port description that indicate the inputs and outputs of the function. But a programmer wants to express an event in a simpler fashion where state variables can just be indexed at ‘neighbor’ offsets and referenced by name in the transition function. SIMP makes this easy to do.

Rather than making the programmer learn a little language just for writing transition functions, SIMP leverages Python’s expressive syntax, providing a means for users to express event transition functions using Python functions. The function implicitly specifies the transition function’s inputs and outputs and their local (neighbor) offsets. At declaration time, an event gives the programmer a means to parameterize the transition function. All of this front-end, necessitates some back-end complexity. A STEP must analyze, validate, and recast the Python function into a form suitable for implementing an event

Syntactically, STEP transition functions are Python functions, but their interpretation is different than that of ordinary Python functions:

- An event applies the transition function in parallel at all sites.
- Signal coordinates are site-relative; state variables without explicit coordinates implicitly reference those of the site being updated.
- Unlike a Python function, a STEP transition function does not have global variables as such, rather the declaration-time *values* of the Python function *parameterize* the transition function—globals having state variable values become transition function inputs and outputs; all others become direct value references.

Because of these differences, the SIMP must analyze and validate the function and transform it that's suitable for a STEP to implement. This includes

- compiling a 'port description' for the Rule's local inputs and outputs and neighbor offsets,
- transforming the transition function to a form that has explicit input parameters and output return values,
- substituting global variables with the current values referenced in their scope,
- raising an error if the transition function tries to modify a global value.

Using the Python introspection methods, SIMP employs a variety of techniques to accomplish these tasks. They include examining the compiled structure of the function, constructing an abstract syntax tree (AST) for the transition function, walking the tree to get information, and transforming it. The end result is an input/output port description and a version of the transition function with explicit input arguments and return values for each input and output. This version is suitable filling in the `input`, `output`, and `function` members of a STEP API `Event` class instance:

`Event`

`generator`	`G`, generator matrix
`input`	in, list of n signal/offset tuples
`output`	out, list of signal/offset tuples
`function`	f, function with n arguments returning a tuple of m outputs

SIMP must perform some analysis to determine the inputs and outputs of transition functions and convert them into a form that's usable by a STEP. Doing so is not trivial. It requires analyzing the structure of the function and the globals that it reads and writes.

But the benefit is that it saves the programmer the considerable pain of having to create port descriptions and transition functions that explicitly declared parameters and return values corresponding to the port description.

The difference is between writing something like

```
def parity():
  c._ = c[0,0]^c[-1,0]^c[0,1]^c[1,0]^c[0,-1]
parity = Rule(parity)
```

and something like

```
def parity(c,c_N,c_E,c_S,c_W):
  c = c_N^c_E^c_S^c_W
  return (c,)
parity_rule = Rule(parity,
                   inputs=[(c,[0,0]),(c,[0,0]),(c,[-1,0]),
                           (c,[0,1]),(c,[1,0]),(c[0,-1])],
                   outputs=(c,[0,0])
                  )
```

Fortunately, with certain conditions and constraints, SIMP has methods to analyze python functions like the one above.

We now describe a methodology for doing the requisite analysis on transition functions. The cleanest way to do it is to analyze the code of a function using its abstract syntax tree (AST) which describes the logical structure of the transition function. SIMP uses AST find to find a transition function's input and output signals, check some of the constraints, and strobe global values. It transforms the original AST into a new AST and compiles a new Python bytecode version that can be used to evaluate the transition function in the inner loop of a scan routine. The new version not only implements the proper semantics, but it also is *faster* and more suitable for doing other things like generating a LUT from the transition function.

The SIMP module for doing these transformations, `simp.rule_analysis`, uses the Python libraries `inspect`–to get the text of the function—and `compiler`—to convert it into an AST and generate Python bytecode from an AST.

Before describing the function analysis and transformation steps, we demonstrate them on a concrete example—the 1D `majority` function below.

```
thresh = 1
def majority():
  sum = a[-1]+a[1]
  if sum>thresh:
    a._ = 1
  elif sum<thresh:
    a._ = 0
```

`thresh` is a global variable that will be replaced by a constant—the value `1`. Assume that `a` is a `Signal`. The only output is `a` (shorthand for `a[0]`). The obvious inputs are `a[-1]` and `a[1]`. But there is yet another input. Because output value of `a` is not set in all cases (i.e. if `sum==thresh`), `a[0]` may be needed to supply the default output value for `a`. Recall that, as discussed near the end of Chapter 4.1.2, the default output value is identity—therefore, the implicit first line of the function is `a._ = a`. If `a._` is assigned in all cases, this implicit first line is not needed and the implicit input `a[0]` can be eliminated.

One of the things the re-compilation handles is default assignment values for transition function outputs. If an output does not have a value assigned in all cases, the default output value is the previous value of the state variable. It defaults to identity. In order to implement this, the output must also become an input to the transition function. So, if the transition function does not assign an output value in all cases, SIMP will add a default assignment. If the output is not already an input, SIMP will add it to the set of inputs.

Fig. 6·1 graphically depicts the AST for `majority` generated by `compiler` (it has been simplified a bit by eliding the `elif` statement and removing a few node attributes). When SIMP performs code analysis, it walks through the tree to find the global names. Names that don't refer to `Signal` objects are replaced with values; notice that the node `Name("threshold")` is replaced by the constant value, `Const(1)`, in the transformed AST of Fig. 6·2.

References to signal objects are also given *proxy names* according to the order in which they occur for the first time as an input or an output. The first input reference, `a[-1]` is given the name `_in0_` while `a[1]` is `_in1_`. This transformation is also shown in Fig. 6·2.

To accomplish the transformation, `simp.rule_analysis` runs a series of postorder visits to the tree. Recall that a postorder visit visits nodes from the bottom up from left to right. The first visit gets global names—they are simply the names that are referenced, but not assigned in the function. The second replaces non-signal, globals (`Name` nodes) with constants (`Const` nodes). The third eliminates constant sub-expressions—for example, reducing `Sum(Const(2),Const(5))` to `Const(5)`. The reduction not only makes the code run faster (which is nice if we

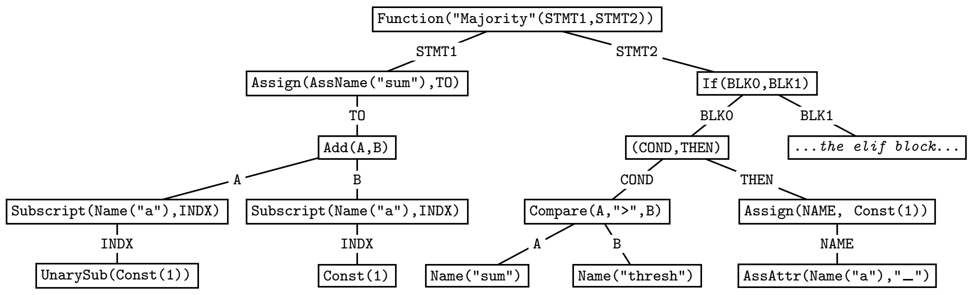


Figure 6·1: Abstract syntax tree the majority rule.

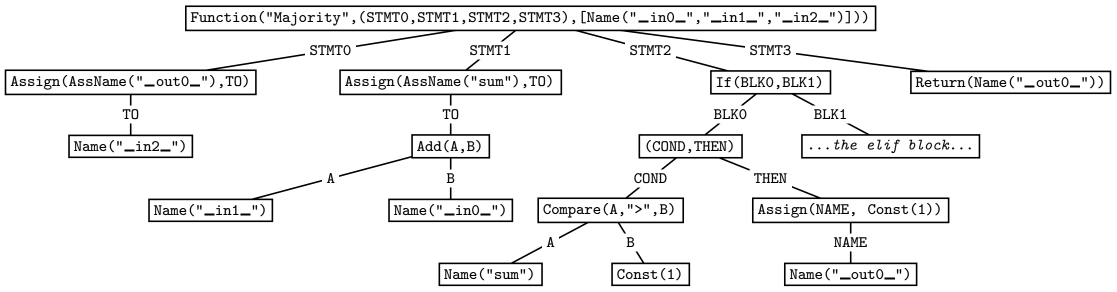


Figure 6·2: AST for majority after strobing and input/output analysis

use the code to evaluate the local function or generate a LUT) but also is useful when we want to obtain subscripts for signals that are given by expressions (`c[2+1]` would be converted to `c[3]`). The fourth visit replaces output signals by their proxy names by looking for dot underscore attribute assignments, while the fifth visit replaces input signals with their proxy names[3].

As signals are replaced with proxy names, a mapping between proxy names and the signal/neighbor-coordinate pairs to which they refer is constructed. All signal

[3]One might think that the fourth and fifth passes could be combined; however, they can not. This is because the local structure of AST subtrees of both input and output references are basically the same. The only difference is that in the case of an output, the output attribute (dot underscore) is assigned. Therefore, the code first replaces the outputs and then the inputs.

references are converted to such pairs and each distinct pair is assigned a distinct proxy name.

Once the AST analysis is complete, the `compiler` module can generate the bytecode for the modified transition function. The bytecode declares the modified transition function. By sourcing the bytecode in a new namespace (empty Python dictionary), SIMP effectively creates a new Python module that contains an executable version of the modified transition function. For `majority`, the modified function is basically

```
def majority(_in0_,_in1_,_in2_):
  _out0_ = _in2_
  sum = _in0_+_in1_
  if sum>1:
    _out0_ = 1
  elif sum<1:
    _out0_ = 0
  return (_out0_,)
```

`simp.rule_analysis` provides a listing of input and output name/value pairs—in this case `{(a,[-1]),(a,[1]),(a,[0])}` and `{(a,[0])}`—so that the STEP implementation knows what values and order of the inputs passed to and outputs returned by this function.

6.1.5 Renderers

SIMP's built-in rendering classes leverage STEP's rendered read operations. The renderers take a rendering `Event` and one or IoSignals as an argument. The IoSignal objects give the values for the red, green, and blue color channels of the image to be rendered and the EVENT indicates how to derive the local values of the colors from the state variables. The renderer classes use the event to construct a "read operation". A rendered read is a sequence consisting of an `Event` that writes to

IoSignals followed by a `Read` operations that reads signal values.

The SIMP renderers include a 2D renderer suitable for rendering two dimensional dynamics or slices of higher dimensional dynamics and a space-time renderer suitable for displaying the space-time history of one-dimensional dynamics. The renderers have an interface designed to integrate seamlessly with SIMP's built-in user-interface class, the console. This console class can be used in a SIMP application to display rendered images on-screen and accept interactive user key-commands for updating the dynamics and issuing user-defined functions.

Renderers are classes that render signals to images. Usually, this involves rendering a set of state values to a RGB array. They do this by constructing the STEP `Read` operations necessary to complete the task and performing any extra buffering that may be needed (in particular, the space-time renderers buffer state information). Renderers are used to obtain 2-D rendered images from signals. In particular, they implement the `Renderer` interface, as such, they are compatible with `Console` objects, which rely on a renderers for generating the images they display. They also provide handy mechanisms for rendering from scripts. Renderer objects implement a common interface so that they may be accessed in a common way by the `Console` class.

6.1.6 The array interface to signal objects

SIMP subclasses the `Signal` class and wraps it an array interface. This interface includes the ability to index array coordinates and get `SignalRegion` objects. The `SignalRegion` objects encapsulate the notion of a `Signal` plus a rectangular region within it. The `SignalRegion` class can, itself, be used to read an array of values from a state variable (SIMP automatically declares a `Read` operation when one calls the `value` method of a `SignalRegion` and issues and flushes the operation in order to get the array of values from the region. A `SignalRegion` that references a single site can also be used to represent 'named' neighbor offsets in a SIMP transition function.

Assigning data or arrays to a `Signal` also implicitly flushes the state and issues a `Write` operation on the signal.

6.1.7 The console, an interactive user interface

The `Console` provides a viewer window and an interactive key command interface for running SIMP programs interactively. The `Console` included with SIMP is built on the `pygame` (`http://pygame.org`) Python interface to the SDL (simple direct media layer).

Chapter 7

STEP Implementations

A central goal of the STEP architecture is to create an abstract CA architecture that can be implemented in a number of different ways. This chapter presents some ways of implementing STEP in software for ordinary computers. We focus on software implementations, because, as opposed to hardware implementations, software implementations are of ubiquitous utility.

Implementing a STEP primarily entails

- implementing state variables, including a storage and I/O access strategy,
- efficiently implementing the CA update event operations.

These aspects are the most interesting because they have the greatest impact upon performance and are the most complex parts of a STEP implementation. Other concerns include implementing the shift operation, implementing stir operation, and performing the necessary bookkeeping to keep track of the current coset locations of the state variables and events. All of these things can be done in a rather straightforward fashion that does not depend too much on the actual architecture or implementation.

We discuss implementation aspects in the context of the four STEP implementations we developed and present the STEP implementations in the order that we developed them. They STEPs are

- `Reference` — A pure-Python reference implementation written for correctness and clarity first and performance second. It defines the behavior of a STEP and is the functional yardstick against which all other STEP implementations are measured. As such, it forms the basis for our cross-validation strategies (see Section 8.3). A STEP is valid if, running the same application on it and the Reference STEP produces exactly the same result at each site and after each operation.[1] Some aspects of the Reference STEP are relevant to all of our STEP implementations. In particular, the strategy for storing signals and representing shifts are the same for all STEPs and we introduce them here.

- `Pc` — A fast implementation with events implemented by statically compiled `C` scan loops. We introduce the "interrupt interpreter" technique for efficiently updating the volume of a CA and interrupting to handle boundary conditions at pre-computed locations.

- `PcThreaded` — This implementation employs multiple threads for a multi-processor/multicore systems. We describe a very simple data partitioning strategy in which separate threads update disjoint slices of the space.

- `PcCodeGen` — An implementation that dynamically generates the scan loops that implement events. It requires a `C` compiler. An important feature is that it can convert event transition functions into C procedures. It also optimizes the scan loop by unrolling the internal loops that read input and write output values.

Because each successive implementation extends the previous, we present them in the order given above.

[1]The only exception is applications that include the shuffle operator, which for performance reasons, different STEPs may implement in a different ways.

7.1 The Reference STEP

Before defining more optimized STEPs *we deemed it prudent to write a reference implementation*, called Reference, *that's easy to verify and debug.* For simplicity, we coded it in pure Python.

We start with the Reference STEP because it is conceptually simple and has aspects that are common to all other implementations. These common aspects include the way it represents shifts and stores signal values. Further, the strategy it it takes performing event scans has some similarity and we take the opportunity in this section to discuss the requirements of the event scan loop.

7.1.1 Shifts: moving data without movement

A shift operation uniformly translates in space the values of a signal by some amount given as a vector. The most straight-forward way to implement a shift is by physically moving values in memory. But, this is actually not necessary. Because a shift really just changes a signal's frame of reference, the implementation need not be physical at all. This realization—that one can implement the logical equivalent of a shift without physically moving data—was one of the major innovations of the CAM-8.

Instead of physically moving data the Reference STEP just shifts the signal's frame of reference. Similar to the CAM-8, the Reference STEP employs logical *offset* vectors that, for each signal, indicate the logical coordinate of the origin of the signal's physical memory.

To shift the frame of reference, a shift operation need only increment an internal offset vector $\mathsf{o}^{<s>}$ that the Reference STEP defines for each signal s. Effectively, a shift of v is just implemented by incrementing $\mathsf{o}^{<s>}$

$$\mathsf{o}^{<s>} \leftarrow \mathsf{o}^{<s>} + \mathsf{v}$$

The Reference STEP ensures that all other STEP operations—events, reads, writes etc.—respect a signal's current offset vector when resolving the physical locations of the state variables. Internally, signal reference $s[\mathsf{x}]$ is converted to $s[\mathsf{x} - \mathsf{o}^{<s>}]$.

$$s[\mathsf{x}] \mapsto s[\mathsf{x} - \mathsf{o}^{<s>}]$$

7.1.2 Signals

Because all of our other implementations inherit the same strategy, we take some time to explain how the Reference STEP implements signals.

Storing signals in `numarray` objects

The storage for a signal is kept in internal `numarray` objects. Read and write operations are accomplished by indexing, slicing and copying data from these internal objects. The `numarray` representation is convenient because the numarray library allows one to either access the multidimensional data directly as a memory buffer in `C` or to access it using convenient Python indexing and slicing methods. The C methods are faster while the indexing and slicing methods are convenient. Because slicing just results in array references, rather than actually copying data, slicing actually is efficient when the volume of the slice is large.

When allocating a signal the Reference STEP allocates two arrays—one for the current state and one to hold the next state that results from applying an event. This is necessary because our STEP implementations employ a *double buffering* strategy to implement events. (The output is written to a special output array and the input value is preserved during the scan. When the scan completes, the output array becomes the present state and the input array will become the next output array buffer.) Therefore, each signal actually has two arrays—one for the current state and one for the next state.

Types

The types of the `numarray` class are compatible with those of the STEP API. The only type that doesn't map exactly into a `numarray` type is the parameterized `SmallUInt` type. The Reference STEP stores signals of this type in an 8-bit unsigned integer (`UInt8`) numarray objects. Like the other implementations, the Reference STEP uses one array per signal.

In order to save storage space, one might consider making the optimization of packing the bits of state from multiple `SmallUInt` signals into a single array. While this optimization does save storage space and reduce the size of the working set during a scan, but it also requires that the implementation shift and mask bits for all the state-variable accesses. While the improvement in storage could be as much as a factor of 8 and reducing the size of the storage would tend to reduce the size of the working set in a scan, in preliminary tests, we found that the performance improvement was not too significant. Therefore, we abandoned this approach, but it might be worthwhile to revisit it in a future STEP implementation should the size of the working set become an issue.

About `numarray` objects

Because the numarray objects are so important to the overall STEP API, they are the format employed for all state variable I/O, we take a moment to define their storage format.

A `numarray` array instance is a multidimensional array. An n-dimensional numarray A is indexed by vectors i such that $0 \leq \mathsf{i} < \mathsf{z}$ where z defines the array's (rectangular) shape. Each array element $\mathsf{A[i]}$ is a variable of type $\mathcal{T}$. In short

$$\mathsf{A[i]} \in \mathcal{T} : 0 \leq \mathsf{i} < \mathsf{z}$$

In their internal storage format, numarray objects are just memory buffers with some special indexing conventions. The number of elements in the array is equal to the product of the elements in the shape vector z. The size of the buffer is at least $(\prod_{i=0}^{n-1} z_i)|\mathcal{T}|$ where $|\mathcal{T}|$ is the size in bytes of the array's element type.

An n-dimensional numarray A is defined by a rectangular shape z, a type $\mathcal{T}$, a one-dimensional memory buffer a that stores the state, and a, yet unmentioned, stride vector t that maps array indexes to memory locations in the buffer. The stride vector indicates how one determines the address of an element i. In order to reference the value of an array element $\mathsf{A}[\mathsf{i}]$ via a multi-dimensional array index i, one must determine the scalar memory index i into the buffer a of the element $a[i]$. This can be computed by taking the dot product of the stride vector t and i. We denote the dot product as $<\mathsf{i}, \mathsf{t}>$. In particular,

$$i =< \mathsf{i}, \mathsf{t} >= \sum_{k=0}^{n-1} i_k t_k$$

and

$$\mathsf{A}[\mathsf{i}] = a[< \mathsf{i}, \mathsf{t} >].$$

Mapping coordinates to array indexes

When signals are allocated on the grid, the mapping from a signal coordinate x to an array index is fairly direct—all one must do is adjust for the offset vector $\mathsf{o}^{<s>}$ as in

$$s[\mathsf{x}] \mapsto A[\mathsf{x} - \mathsf{o}^{<s>}]$$

where $\mathsf{x} - \mathsf{o}^{<s>}$ is taken modulo the dimension vector d that defines the bounds of the coordinate space.

In contrast, to this simple scenario, things are more complicated when signals are allocated on a proper sublattice of the grid. The logical coordinates and the physical array indexes cannot be one and the same. One must have a strategy for mapping one to the other.

The simplest strategy would be to just allocate an array with the same size as the grid and only use a subset of the values—the ones corresponding to the sublattice. This, however, is not very space-efficient, especially when the sublattice is sparse. The more storage-efficient strategy is to allocate an array that's only as large as necessary for storing each signal site. This is what the Reference STEP does.

The approach that the Reference STEP takes for mapping the sites of a signal into an array is based on the same signal region-to-array mapping strategy as was described in Chapter 5.1.4. In particular, for a space of size d, signal-to-array mapping is exactly the one that one would get by selecting the signal region from the origin up to d, that is $\mathcal{R}(0, \mathsf{d})$. Geometrically, can think of the space as being divided up into an orthogonal tessellation defined by a signal's lattice's rectangular unit cell $\mathcal{U}(\Lambda)$. Each rectangle corresponds to one array element and that element contains the value of the signal (sublattice) site that falls within it.

More concretely, the storage strategy is as follows. For a signal with a lattice defined by the HNF generator G, the Reference STEP allocates an array of size d/g where g is is the diagonal of G and / denotes element-wise division. The type of the array is the same as that of the signal. The array index i of a site at coordinate x is

$$\mathsf{i} = \lfloor \mathsf{x}/\mathsf{g} \rfloor$$

and

$$s[\mathsf{x}] = \mathsf{A}\left[\ \lfloor \mathsf{x}/\mathsf{g} \rfloor\ \right]$$

Or, taking the offset into account,

$$s[\mathsf{x}] = \mathsf{A}\left[\ \lfloor (\mathsf{x} - \mathsf{o}^{<s>})/\mathsf{g} \rfloor\ \right]$$

7.1.3 Events

The Reference Step takes a pure-Python strategy towards implementing events. Nevertheless, the implementation has some interesting aspects that are useful for understanding the other implementations. Before going into more detail about how it computes an event, let's take a moment to consider what a CPU must do in general to implement an event.

A generic event scan loop

As described in the STEP chapter, Chapter 5, an event performs the following computation

$$\text{out}[\mathsf{x}] \leftarrow f(\text{in}[\mathsf{x}]) : \mathsf{x} \in \Lambda + \mathsf{c}$$

Conceptually, there's nothing too complicated about implementing this on a CPU. One can just perform a scan over all sites x in the event, read local inputs (in[x]) and map them to local outputs (out[x]). At a high level, the CPU just needs to perform the following loop

```
for each site in the event's lattice do
    1) read the $n$ local inputs
    2) compute the outputs as a function of the inputs
    3) write the $m$ local outputs
end for
```

In more detail, in order to perform an event e that has a lattice $\Lambda^{<e>}$ and a current coset position $\mathsf{c}^{<e>}$, a STEP must implement a loop of the following sort

```
for $\mathsf{x}$ in $(\Lambda^{<e>} + \mathsf{c}^{<e>})$ do
    for $i = 0$ to $n - 1$ do
        inputs[i] $\leftarrow s^{<\text{in}_i>}[\mathsf{x} + \mathsf{v}^{<\text{in}_i>}]$
    end for
    outputs ← $f$(inputs)
    for $i = 0$ to $m - 1$ do
      $s^{<\text{out}_i>}[\mathsf{x} + \mathsf{v}^{<\text{out}_i>}] \leftarrow$ outputs[i]
    end for
  end for
```

where ‘inputs’ is a vector of the n local input values of the event and ‘outputs’ is a vector of the m local output values of the event.

There are three interesting parts in this loop—generating the site index x for the event, dereferencing memory locations of signal values $s[\mathsf{x} + \mathsf{v}]$, and computing the transition function. Efficiently implementing an event requires that one efficiently generate the indexes, dereference memory locations in an order that is efficient for the memory subsystem (i.e. the cache) and implement the transition function in as few operations as possible. The Reference STEP does not really attempt to optimize any of this—that’s something that, as described in Section 7.2 and Section 7.4, the other STEP implementations will do.

Nevertheless, the Reference STEP does need a strategy for 1) generating the scan indexes x, 2) dereferencing the state variables $s[\mathsf{x}+\mathsf{v}]$, and 3) implementing the transition function f.

Generating the scan index

Generating the scan index is a relatively simple matter. We just iterate over the sites in the event’s lattice. Now, given an event with a generator G having a diagonal of g and a space of size d, we generate scan indexes $\mathsf{w} : 0 \leq \mathsf{z}$ where z is the ‘index space size’ and defined as $\mathsf{z} = \mathsf{d}/\mathsf{g}$ with the division operator ‘/’ taken element-wise. The scan indexes are generated by incrementing incrementing w as a multi-dimensional counter with a radix given by z. The following computes the multi-dimensional increment in n-dimensions

```
$i \leftarrow n-1$
$w_{n-1} \leftarrow w_{n-1}+1$
**repeat**
    $w_{i-1} \leftarrow w_{i-1}+(w_i/z_i)$
    $w_i \leftarrow \mod(w_i, z_i)$
    $i \leftarrow i-1$
**until** $i=0$
$w_0 \leftarrow \mod(w_0, z_0)$
```

(Note that the the division (w_i/z_i) is integer division.)

Now, to generate x from w we simply use the following

$$\mathsf{x} = \mathsf{wG} + \mathsf{c}$$

where c is the event's current coset position.

Dereferencing a signal

We must translate a signal coordinate reference $s[\mathsf{x}+\mathsf{v}]$ to an array index reference $\mathsf{A}^{<s>}[\mathsf{i}]$ where $\mathsf{A}^{<s>}$ is the array for s. The expression for computing i is

$$\mathsf{i} = \mod([(\mathsf{x}+\mathsf{v}) - \mathsf{o}^{<s>}]/\mathsf{g}^{<s>}, \mathsf{z}^{<s>})$$

where all vector operations are element-wise, and $\mathsf{o}^{<s>}$ is the offset vector, $\mathsf{g}^{<s>}$ is the generator matrix diagonal, and $\mathsf{z}^{<s>}$ is the array size of the signal s. (Recall that the array size is defined as $\mathsf{z}^{<s>} = \mathsf{d}/\mathsf{g}^{<s>}$ where d is the size of the space.)

Implementing the transition function

In the Reference STEP, implementing the transition function is a simple matter—it just directly applies the Python function passed to it by the STEP API. The input vector forms the set of arguments while the output vector forms the set of outputs. Because calling an interpreted Python function may be slow, the other implementations all optimize this by compiling the transition function into a lookup-table or a C procedure.

7.1.4 Double buffering

When a signal appears as both an input and an output, it can not, in general, be written back in-place. This is because other yet-to-be-scanned sites may actually require the original input value. Therefore, input values must buffered until all sites that need to read them have been scanned.

The simplest buffering strategy is called *double buffering*. With double buffering we keep two arrays, one for the current-state input values of the signal and one for the next-state output values. After the scan completes, the output array becomes the current state array. Although in comparison to alternatives that minimize the amount of buffering, double buffering may double the storage size and memory bandwidth requirements for a scan, it is much simpler and in our tests did not incur much performance cost.

However, one implication of double buffering is that it may, when an event is defined on a sublattice of a signal that it writes, require that the input be pre-copied into the output array. Here's why. An event defined on a sublattice of a signal may not actually write all of the values in that signal—the event may entirely skip certain cosets of the signal. In this case, the next-state values for those skipped cosets should actually be the present-state values. In order for those present state values to appear in the output array, our STEP implementations copy the input array to the output array and then perform the scan. This way, the scan can just overwrite the values in the output coset of a signal and inherit from the input the values in the cosets that it does not write.

7.1.5 Stirring

The implementation of the stir operation is simple, but quite efficient. Stirring a signal or event randomizes its coset position and, in the case of a signal, rearranges its state variable values.

To randomize the coset position coset c the Reference STEP selects a new c in the range $0 \leq \mathsf{c} < \mathsf{g}$ where g is the generator for the signal or event.

To rearrange the data our STEP implementations just treat the multi-dimensional array A of a signal s as one-dimensional buffer a and perform a circular shift on this buffer. If the buffer's size is $|a|$ our STEPs select a random cut point c such that $0 \leq c < |a|$. Using the cut point, it performs a circular shift,

$$a'[i] \leftarrow a[\text{mod}(i + c, |a|)] \quad : \quad 0 \leq i < |a|$$

Using numarray, this is easy to do via the following two slice assignments:

$$a'[|a| - i : |a|] \leftarrow a[0 : i]$$

and

$$a'[0 : |a| - i] \leftarrow a[i : |a|].$$

7.2 The Pc STEP

Let's now consider a faster implementation, the Pc STEP. With the exception of events, most of the Reference STEP operations are actually fairly fast. Therefore, Pc STEP implements all of its operations, except for the event in the same way as the Reference STEP. The Pc STEP optimizes the implementation of events by compiling the transition function into a LUT and implementing the event scan loop in an optimized fashion.

The Pc STEP implementation is optimized for a common case in CA applications where state variables have a small number of bits (e.g. they are `SmallUInt` types) and events have small numbers of inputs. It compiles event functions into lookup tables and can thus implement the function with a single lookup.

The scan loop is optimized for scanning sites row-by-row. This not only orders memory access in a way that optimizes performance, but also allows the data pointers for the inputs and output to simply be incremented most of the time. The only exception to this is when the scan reaches the end of a row and the pointers need to wrap-around or the scan needs to move to the next row. To handle these situations the Pc STEP it compiles scans into *interrupt programs* that control the running of a tight scan loop written in `C`.

When the scan interpreter reaches the end of a row and needs to proceed to the next row or a signal in the row wraps around, the scan interrupt program instructs the scan loop to interrupt. When it interrupts, the tight scan loop drops into an *interrupt interpreter*. The interrupt program instructs the interpreter to update the pointers of the variables that caused the interrupt. It moves them to the next row or makes them wrap-around. After doing this, the interrupt program sets a special *interrupt counter* to the number of loop iterations until the next interrupt. This way, in most iterations, the scan loop need only incur the cost of decrementing and checking the interrupt counter.

7.2.1 Representing the transition function with a LUT

A LUT is compiled by evaluating the transition function with all possible inputs inputs and recording the corresponding outputs. The LUT is indexed by a binary vector containing the input values. To minimize the width of the LUT, outputs are also packed into binary vectors. The cost of doing this is that, when a lookup is performed, output values must be unpacked using bitwise mask and shift operations. The benefit is that the lookup table can be made smaller.

Caching

Because a LUT may be expensive to compile, the PC implementation uses a persistent cache to hold LUT data. This way, the LUT need only be recompiled when a new transition function is presented to the system. This is important because larger LUTs may take minutes to compile. To cache a LUT for a transition function, we must generate a unique key. The key we use is the strobed AST for the transition function together with the types (and sizes) of the inputs and outputs. This way, if the function changes, the LUT will be recompiled.

7.2.2 The high cost of computing site indexes

We now take a moment to analyze the cost of computing a site index and use this cost to motivate the strategy that the Pc STEP takes towards implementing a scan. As we've already seen in the Reference STEP's scan description, in order access a site, we must perform a rather complicated bit of vector arithmetic to bring an index vector down to an element address. In particular, we must translate a signal coordinate reference $s[\mathsf{x}+\mathsf{v}]$ to an array reference $\mathsf{A}^{<s>}[\mathsf{i}]$. In the Pc STEP, the expression for computing computing i is

$$\mathsf{i} = \mathrm{mod}(((\mathsf{x}+\mathsf{v}) - \mathsf{o}^{<s>})/\mathsf{g}^{<s>}, \mathsf{z}^{<s>})$$

where x is the scan site, v is the neighbor offset, $\mathsf{o}^{<s>}$ is the signal's current shift offset, $g^{<s>}$ is the diagonal of the signal's generator matrix, and $\mathsf{z}^{<s>}$ is the size of the array ($\mathsf{d}/\mathsf{g}^{<s>}$ where d is the size of the space) . Going even a bit farther, we must resolve the multi-dimensional index i to a memory index i with

$$i = <\mathsf{i}, \mathsf{t}>$$

where t is the stride vector for the array $\mathsf{A}^{<s>}$.

Left as-is, this would be *very expensive* to compute for each site in a scan. The cost of performing of this computation—$4k$ integer additions/subtractions and $3k$ integer multiplications/divisions/modulo operations, where k is the number of dimensions. This is unacceptably high because it must be performed $n+m$ times—once for every input and output.

The cost of this completely dwarfs the rest of the cost of updating a site—especially when the transition function is implemented as a LUT. The remaining cost is that of actually performing the buffer reads and writes (e.g accessing $a[i]$) and applying the local transition function. Even a single integer division can be many times more expensive than computing the transition function.

7.2.3 Performing scans row-by-row

Fortunately, by performing the scans one row at a time, one can avoid having to pay a high per-site cost to resolve memory array indexes. Most of the time, the cost can be made as small as simply incrementing the array index i. To see how this can be the case, consider that, when we scan within a row, the buffer index usually just needs to be incremented to the next site within the row. The only exception to this is when it needs to wrap-around or the scan needs to move to the next row.

In particular, within a row, a given a signal index, i, usually need only be incremented by a static amount determined by the signal's array stride and the stride of the event within the row. In particular, the amount of the increment is

$$\mathrm{inc} = t_{n-1}(g^{<e>}_{n-1,n-1}/g^{<s>}_{n-1,n-1})$$

where t_{n-1} is the array stride in the least-significant dimension and $(g^{<e>}_{n-1,n-1}/g^{<s>}_{n-1,n-1})$ is the number of signal sites the event strides through as it increments the scan index within a row. The only exceptions to these uniform increments occurs when a signal wraps around or the scan moves on to the next row.

If it weren't for these exceptions, one could just maintain a pointer for each input and output and increment it by the appropriate amounts between sites. In this case, the cost of indexing becomes simply that of a single scalar addition for each input and output. This is about as inexpensive as the indexing can get.

Unfortunately, the exceptions break the symmetry of the scan. The simplest way to deal with this break in the symmetry is to just compute the array index afresh for each site like the Reference STEP does. However, this is very expensive.

A better strategy, is to pay the cost of computing the base pointer within a row once per row and then, within a row, to add an offset to this base pointer to resolve the address of the values needed as an input or output. The offset can be taken modulo the size of the row in order to implement wraparound. The following pseudocode demonstrates

```
offset[i]=(offset[i]+inc)%row_size;
ptr[i] = row_base[i]+offset[i];
```

Unfortunately, this requires a modulus operation for each input and output. At this point, one can either pay the cost of performing the modulus operation, which is equal to the cost of 9 additions. In the special case that the row width is a power of 2, the modulus can be taken my masking the value and this option becomes quite attractive. (Note that the CAM-8 took a related approach in hardware.)

Fortunately, it is not necessary to restrict the size to a power of 2 and we can achieve level of performance that's better by generating *interrupts* at the locations where a state variable needs to wrap-around or move on to the next row.

7.2.4 An interrupt-based scan approach

The interrupt-based scan approach is to perform a scan row-by-row and interrupt, adjusting input and output pointer values only at those points when a signal needs to wrap-around or the scan needs to move on to the next row. The flow chart in Fig. 7·1 depicts the flow of such a scan. Normally, the scan just runs the scan loop that reads the inputs, evaluates the transition function, writes the outputs and decrements the interrupt pointer. However, when the interrupt counter reaches zero, we drop into the interrupt interpreter. The interrupt interpreter modifies any pointers that need to wrap-around or move to the next row. It then sets the interrupt counter to the number of sites to be scanned before the next interrupt.

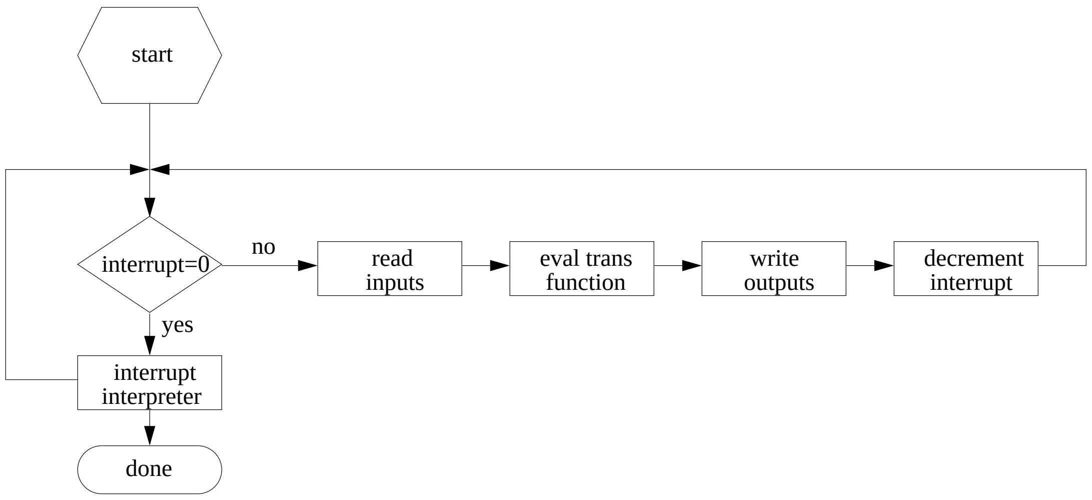


Figure 7·1: Performing an interrupt-driven scan

Consider a scan that computes a one-dimensional event defined by `parity1d`

```
def parity1d():
  c._ = c[-1],c[0],c[1]
parity1d = Event(parity1d)
```

The input set is in $= \{(c, [-1]), (c, [0]), (c, [1])\}$ and the output set is out $= \{(c, [0])\}$

The Fig. 7·2 depicts the interrupts in a scan of a space of size X for this rule.

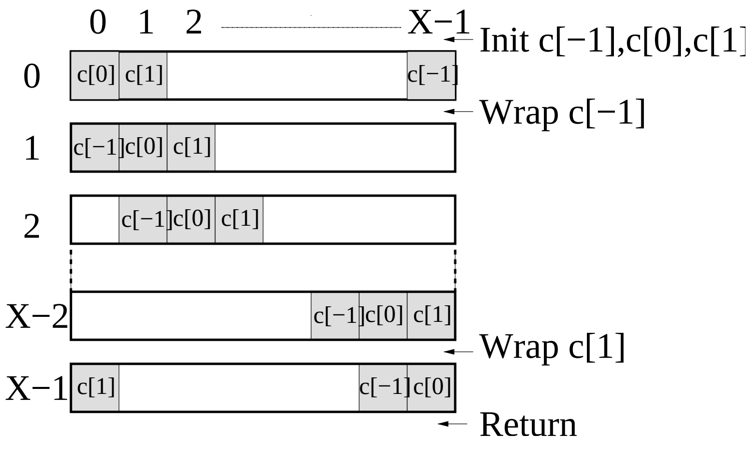


Figure 7·2: Interrupt based scan example

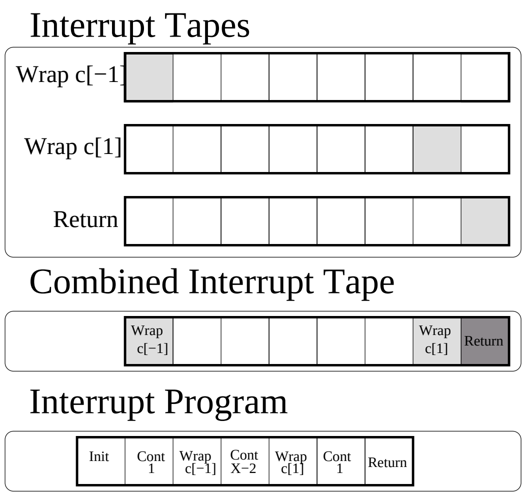


Figure 7·3: Interrupt Tapes

The locations where these interrupts occur can be precomputed. One strategy would be to precompute an interrupt array for each pointer that indicates whether an interrupt had occurred and the new value for the pointer when it does occur. This would require each pointer to check its interrupt array before incrementing to see whether it needs to jump to a new location. This is the “interrupt tapes” strategy that Fig. 7·3 depicts. There is a tape that indicates the location of the interrupt for the input `c[-1]`, one that indicates the location if the interrupt for `c[1]` and one called "Return" that indicates the interrupt for the end of the scan. When this is triggered, the scan routine should return.

Even when there are no interrupts, this method costs n interrupt checks per iteration—n is the number of pointers—rather than n modulus operations. Over a scan of m sites, the cost is approximately nm times the cost for checking for interrupts plus the number of interrupts times the handling cost per interrupt.

The cost of interrupt checking per iteration can be made constant by making a combined interrupt array that indicates whether any interrupt has occurred and what kind of an interrupt it is. In this interrupt handler becomes more complicated in that it must determine which interrupt has occurred. Fig. **??** depicts this with the "combined interrupt tape" .

When we precompute the interrupts in this way, we are essentially creating an *interrupt program* that controls the scan pointers. The interrupt program is run by the scan code and controls the interrupt handler. In our current description, the interrupt tape is as long as the array.

Note that the interrupt tape has long stretches with no interrupts. These stretches can be run-length encoded. Rather than checking an array to determine whether an interrupt has occurred, the scan can keep a register that encodes the number of iterations until the next interrupt. It is decremented once per iteration and an interrupt occurs when it reaches zero. When an interrupt occurs, the handler must figure out which interrupt has occurred, service it, and set the next value for the interrupt counter—all of which can be encoded in the interrupt program. The only cost of doing this is that the interrupt program generator must precompute the location of the next interrupt.

In the course of our discussion, the interrupt program has become more complex, but it has also become shorter. The interrupt handler has become something of an interpreter that runs the interrupt program.

7.2.5 The interrupt-driven scan loop and scan interpreter

We now detail the scan loop that PcCodeGen employs to perform a scan. This loop first checks to see whether an interrupt has occurred, and if it has, drops into an interrupt interpreter that adjusts the pointers that caused the interrupt. If it has not occurred, the scan just increments the pointers to the current values. When the program drops into the interrupt interpreter, it stays in the interrupt interpreter until the interpreter sets the interrupt_counter to a non-zero value or returns.

The scan-loop is the following

```
interrupt_counter $\leftarrow$ 0 {Start with an interrupt}
**while** 1 **do**
  **while** interrupt_counter=0 **do**
    execute instructions in the interrupt interpreter
  **end while**
  input $\leftarrow$ 0 {initialize output array to zero}
  **for** $i = 0$ to $n - 1$ **do**
    input $\leftarrow$ input|(mem[ptr$^{<\text{in}_i>}$] << shift$^{<\text{in}_i>}$) {read input values into input bit vector}
    ptr$^{<\text{in}_i>}$ $\leftarrow$ ptr$^{<\text{in}_i>}$ + inc$^{<\text{in}_i>}$
  **end for**
  output_ptr $\leftarrow$ lut_ptr + inputs << lut_shift {perform LUT lookup/compute transition function}
  **for** $i = 0$ to $m - 1$ **do**
    mem[ptr$^{<\text{out}_i>}$] $\leftarrow$
```

```
            $\text{mem}[\text{output\_ptr} + \text{out\_ofst}^{<\text{out}_i>}]\&\text{mask}^{<\text{out}_i>} >> \text{shift}^{<\text{out}_i>}$
        $\text{ptr}^{<\text{out}_i>} \leftarrow \text{ptr}^{<\text{out}_i>} + \text{inc}^{<\text{inc}_i>}$
    **end for**
    $\text{interrupt\_counter} \leftarrow \text{interrupt\_counter} - 1$ {decrement interrupts}
**end while**
```

The interrupt interpreter maintains its own internal program counter and has an instruction set for modifying the values of variables in the scan loop. In particular, it can modify pointer values like $\text{ptr}^{<\text{in}_i>}$ and $\text{ptr}^{<\text{out}_i>}$. It modify variable values used in the scan loop because of the way that the scan loop is coded. Rather than having normal variables, all the variables in the scan loop are actually *register* references. The registers are the memory addresses that the interrupt interpreter can access. For example, the pointer output_ptr might, in the scan, actually be stored in register number 15 and referenced with the code '`R[15]`'. The following is the instruction set of the interrupt interpreter

```
Instruction set
\begin{verbatim}
ASSIGN a,b         R[a] <- b
CONTINUE           Continue in the loop
RETURN             Exit from the loop
ADD a,b,c          R[a] <- R[b]+R[c]
SUB a,b,c          R[a] <- R[b]-R[c]
INC a              R[a] <- R[a]+1
DEC a              R[a] <- R[a]-1
MOVE a,b           R[a] <- R[b]
SLT a,b,c          R[a] <- R[b]<R[c]
BNZ a,b            if R[b]!=0: pc <- b
JUMP a             pc <- a
BZ  a,b            if R[b]==0: pc <- b
```

So, in order to modify the parameters of the scan, the interrupt program only need assign a value to the appropriate register. The interrupt program generator is aware of the register conventions of the scan loop and compiles the interrupt program accordingly.

Let's examine an interrupt program that implements the 1D scan example from above. In this program, for clarity, we use logical names for the pointers rather than the register values that actually store them.

```
ADDI     ptr_in0  , buf_in0,X-1 // initialize the scan
ADDI     ptr_in1  , buf_in1,0
ADDI     ptr_in2  , buf_in2,1

ASSIGN   interrupt_counter,1    // scan one site

ADDI     ptr_in0  , buf_in0,0   // interrupt to wrap c[-1]
ASSIGN   interrupt_counter,X-2  // scan X-2 sites

ADDI     ptr_in2  , buf_in2,0   // interrupt to wrap c[1]
ASSIGN   interrrupt_counter,1   // scan 1 site

RETURN                          // interrupt to return from the scan
```

In order to generate a scan program like this one, the Pc STEP iterates over each row in the event's lattice. It computes the coordinate x where the row starts and, based upon this coordinate, computes the offset into the storage buffers of each of the inputs and outputs. (It always computes offsets instead of actual memory pointer values so that, as required by the double buffering strategy, the scan program can be employed to run with different array buffers)

In order to get the index x where the event starts in a row, the Pc STEP, similar to the Reference STEP generates a scan index w, but rather than incrementing it in the least significant dimension w_{n-1}, it sets w_{n-1} to zero increments it in the

$n-2$-dimension so that the index jumps from row to row.

To compute the x it uses the following expression for each row index w.

$$\mathsf{x} = \mathrm{mod}(\mathsf{w}\mathsf{G}^{<e>} + \mathsf{c}^{<e>}, [\mathsf{d}_0, d_1, \ldots, d_{n-2}, g^{<e>}_{n-1,n-1}])$$

The modulus in the higher dimensions implements wrap-around. In the least significant dimension, it ensures that the scan starts at the beginning of the row—the modulus selects the site within $g^{<e>}_{n-1,n-1}$ of the origin. This is if we want to start the scan from the beginning of the row as the generator matrix may have skewed generators in higher dimensions.

Given the index x, the scan can compute the array offset for each input/output signal s. It does this for each signal. It can also compute how far into the scan for the row it will be before each input/output needs to wrap around. One it has computed this for each input/output it sorts them in order of where in the scan they occur and generates the interrupt handling code for each interrupt in that order. After each interrupt, it assigns the interrupt_counter to the span of sites until the next interrupt.

Because compiling the interrupt program may be expensive—the Pc STEP must compute the array offsets for each row and for each signal—the cache. The exact interrupt program is a function of 1) the lattices, internal shift offsets, and neighbor vectors of the input and output signals, 2) the lattice and coset position of the event and 3) the size of the space. The event scan interrupt program is cached for unique combinations of these parameters.

The interrupt program will need to be compiled whenever these values change. One implication of this is that, in order to avoid generating a separate interrupt program for each offset position, the Pc STEP should avoid having internal offset vectors that increment indefinitely. The Pc STEP does this this by eliminating the offset vector when writing output values—because the data is all being moved anyway, there's no reason not to eliminate the shift. It also may perform a physical shift of the data if, over a sequence of shifts, a signal moves too far from the origin.

7.3 The PcThreaded STEP

The PcThreaded STEP optimizes the event scan for the case where there are multiple processors. In particular, it allocates a user-specified number of threads as a pool (the initialization parameter `nthread` controls the pool size) and, when performing an event, divides the space up into `nthread` non-overlapping regions and has each thread take care of updating a single region. In doing this, the threads all access the same shared memory for the state variables, but update distinct subsets of it.

This type of implementation is good when there are multiple processors that can be exploited to run each of the threads. Usually, the user will want to set `nthread` to the number of available processors.

The approach that the PcThreaded STEP takes to partitioning the space is very simple. It just divides the space in the most-significant dimension d_0 into blocks of size d_0/n where n is the number of threads. Then, when generating the scan program, it generates n separate scan programs—one for each thread.

7.4 The PcCodeGen STEP

The Pc STEP is much faster than the Reference STEP. However, owing to the fact that it uses a statically compiled scan loop, it is not as fast it could be.

Indeed, the scan loop must loop over the inputs to read them into a bit vector (word) that acts as an input LUT index; it must loop over the outputs to unpack the LUT output bit vector and write it into the outputs. In principle, if the number of inputs and outputs and the number of bits in each were known ahead of time, these read and write loops could be unrolled and compiled into more efficient code. However, because there are many possible combinations for number of inputs, outputs and their respective bit sizes (and hence bit shifts) doing this would require generating and compiling code on-the-fly. This requires a level of infrastructure, namely that of an installed and properly configured `C` compiler, that a casual user may not have. Therefore, we postpone this more optimized implementation for the PcCodeGenStep which builds on PcStep and can generate and compile code on-the-fly. This capability has the added advantage of allowing the implementation to transform the transition function into C and thus implement more complicated transition functions than are possible to implement as a LUT.

Unlike the other STEP implementations, `PcCodeGen` does not solely use lookup tables to implement transition-functions. If the number of bits of input exceeds a predefined threshold of 16 (`maxlutsize` parameter Section 7.4), it uses `C` code instead. This has the advantage of allowing `PcCodeGen` to use a wider variety of types. However, when it must generate `C` code the `PcCodeGen` places some additional restrictions on the transition-functions and will raise an error when a transition-function does not obey them.

Because C requires that types be statically declared, but types are dynamic in Python, it must infer static types for the local variables in the transition function. Because the values of the global variables and the inputs and outputs have, at this point, been resolved, the code generator has a significant amount of type information to use in inferring types for the local variables. To infer the types of the local variables, it follows some simple rules that propagate types on the AST according to the sequence of assignments and operators. The rules are simple, all integer types are represented by an Int32 type, while floats are represented by a Float32 type. Arithmetic operators on a float and an int result in a floating point type.

Once the AST has been decorated with type information, the code generator employs a set of rules for converting Python functions and operators into C.

It's also possible to generate C code on-the fly. Similar to the code analysis used to determine an event's inputs and outputs and to strobe and make constant the global variables of a transition function in, Chapter 6.1.4, the code-generating implementation uses the abstract syntax tree to generate `C` code. Because most of the programming language constructs of `C` and Python are the same, most of the nodes in the AST have a fairly direct to C. When the code generating implementation finds a type for which it can not generate code, it raises an error that indicates why it could not implement the function.

The tricky part is in converting the dynamic types of Python to static types in C. In general, its not possible to do this, so the PcCodeGen implementation places some restrictions on the form of the transition functions and does "type inferencing" to infer static types for the types used in the transition function.

Because the types of the state variables are declared explicitly, the code generating implementation can just use these types directly. In other cases as in construction of intermediate values, the code generating implementation takes the default behavior of declaring all intermediate types as integers or floating point variables based upon the types of the inputs to the assignment.

7.4.1 Limitations on the transition function

We'll now address the extra limitations that the code generating implementation places on transition functions. If any of these limitations are violated, the PcCodeGen STEP will raise an error indicating the issue.

Limited variable types

Because all variables must be converted to simple `C` types, only signals and integer and floating point types can be used in a transition-function. More complicated

Python types such as tuples, lists, and dictionaries are not allowed.

Limited statements

Only basic assignment and `if` statements are recommended. `while` statements will compile, but, because of the possibility of infinite loops and long per-site computations, are not a advisable when they can be avoided. All other statements like `try`, `raise`, and `for`[2] expressions are prohibited.

No sub-function calls

With the exception of the functions `float` and `int`, which are used to cast variables to the floating-point and integer types, a transition-function can not call sub-functions. This is because PcCodeGen does not analyze sub-functions. This restriction may be lifted in the future.

Type inferencing

Unlike Python code, in which a name may hold any type of data, the `C` code variables into which PcCodeGen converts names, must have static types—in particular, one can not store a floating point value in a variable at one time and an integer in it at another. In `C`, type must be declared from the beginning.

However, Python syntax does not support type declaration, so `PcCodeGen` must infer the types of names when converting them to `C` variables. `PcCodeGen` uses types associated with signals and values in the frozen context of the transition-function to infer types for names. If a name is assigned to an integer value, such as `1`, `PcCodeGen` infers an integer type for it; if a name is assigned to a floating-point value, such as `1.1`, `PcCodeGen` a floating point type for it.

[2]The `for` statement is not allowed because it uses list comprehension.

Sometimes the PcCodeGen will encounter conflicting types for a given variable and will resolve them using its precedence rules. For example, in the following code, assuming that `I` is a `UInt16 Signal` and `F` is a `Float32` signal, in the following transition-function

```
def inconsistent():
   if I<4: temp = F
   else: temp = I
   F._ = temp
```

the types of `temp` conflict. PcCodeGen resolves this by casting the type to a `Float32`. Whenever a conflict occurs, it just casts values up to either integers with more bits or floating point numbers which can represent larger values without overflow or underflow.

Assuming that we want `temp` to be an integer, we can rectify this by using the `float` coercion function as in

```
def inconsistent():
   if I<4: temp = F
   else: temp = int(I)
   F._ = temp
```

Most of `Signal` types (`SmallUInt`, `UInt8`, `Int8`, `UInt16`, `Int16`, `UInt32`, `Int32`) map to integer types in the transition-function. Only the `Float32` type maps to floating point.

Chapter 8

Analyses

In this work, our goal is to create a framework for computing with cellular automata that makes CA programs simple to express, yet efficient to implement. The previous chapters have addressed qualitative, theoretic, and implementation aspects of a framework that targets these goals. This chapter analyzes quantitative aspects of the STEP framework.

We are interested to know whether the STEP framework and the SIMP programming environment makes programs simple to express. We've already seen qualitatively that they use high-level constructs to represent things like CA events, state variables, and allocations on sublattices. In this chapter we quantify these contributions in measurable terms such as number of lines of code and hours of programmer effort. It turns out that CA programs in SIMP have sizes that are similar to those in high-level languages like NetLogo and are smaller than those in lower-level hand-coded C and CAM-8 Forth. The development time of SIMP CA programs is much shorter than that of hand-coded C version.

We would like to know whether the implementations of STEP programs are indeed efficient. To analyze this, we compare the performance of CA programs written in SIMP and implemented by the PcCodeGen implementation to that of C. Their performance is almost the same even though the C version took much longer to develop. We also compare the performance of SIMP programs to other high-level implementations such as NetLogo. SIMP programs are orders of magnitude faster.

We also compare the size of the code resulting from various coding strategies in SIMP. In particular, we show that the BPCA written in SIMP is much more compact than previous approaches that employed in LGA to implement BPCA.

We have developed a variety of STEP implementations and in Section 8.2 we compare them. The STEP framework makes comparing the performance of CA programs easy—one can run exactly the same program on both and compare the results. The capability benchmark implementations on the same program in-and-of-itself is a quantitative demonstration of the effectiveness of the STEP framework in decoupling implementations from applications.

The most important performance parameter is the throughput. For a given dynamics, we measure the throughput of a STEP in the millions of sites per second (Msite/sec). This is a good measure because it's independent of the size of the CA space or the number of iterations of the dynamics. It allows us to compute the cost of a space-time volume of CA processing.

We examine performance across a number of applications and a number of dimensions including complexity of the CA events, the style of the program (e.g. whether its a CA, LGA, or BPCA etc, and the size of the space.

We find that moderately large space-time volumes are sufficient to amortize the overhead cost of compiling the dynamics, issuing updates from the API, and handling boundary conditions. The performance achieves that of hand-coded C and the cost per site update is mostly independent of the total volume. However, for small space-time volumes in two dimensions, with only thousands of sites or only a few updates, the overhead is noticable.

The code-generating STEP allows one to increase the complexity of the transition function without hitting the "exponential limit" inherent to LUT based approaches. The results highlight the effectiveness of switching between LUT and compiled code for more complex events with larger number of inputs.

We also demonstrate that the simple data-parallelization strategy of PcThreaded achieves ideal speedup with small numbers of CPUs. The results suggest further optimizations and improvements that might be made to the implementation strategies, which we discuss along-the-way.

8.1 SIMP programs

We base our comparisons on the canonical CA, LGA and BPCA programs that we've already introduced, namely, the Greenberg-Hastings CA, HPP and the Polymer CA. To this, we add Conway's Game of Life, the most popular of the CA dynamics, purely because it is easy to find implementations of it.

We profile the length of several programs written in SIMP and compare them both with similar programs written in C and various CA programming languages, and array programming languages.

8.1.1 Comparison with NetLogo

NetLogo is a programming environment for agent-based computing. It has its roots in the StarLogo software of the MIT Media Lab—software that originally ran on The Connection Machine (Hillis, 1986). NetLogo is optimized for simplicity rather than speed.

As such, it's impressive that the NetLogo version of Conway's Game of Life and the SIMP version both have the same number of lines of code, around 35 lines, without comments. Nevertheless, the SIMP version runs between 10 and 100 times faster than the NetLogo version.

The NetLogo version of the CA update looks like the following (note that there is essentially no difference between the state and the rendered value):

```
to cell-birth
  set living? true
  set pcolor fgcolor
end

to cell-death
  set living? false
  set pcolor bgcolor
end

to go
  ask patches
    [ set live-neighbors count neighbors with [living?] ]
;; Starting a new "ask patches" here ensures that all the patches
;; finish executing the first ask before any of them start executing
;; the second ask.  This keeps all the patches in sync with each other,
;; so the births and deaths at each generation all happen in lockstep.
  ask patches
    [ ifelse live-neighbors = 3
        [ cell-birth ]
        [ if live-neighbors != 2
            [ cell-death ] ] ]
end
```

The SIMP version looks more like procedural programming language code (NegLogo is more functional) and relies upon fewer builtin concepts such such as an implicit neighbor set. The core specification of the dynamics looks like the following:

```
def life():
    sum = p[-1,-1]+p[-1,0]+p[-1,1] + \
          p[ 0,-1]+        p[ 0,1] + \
          p[ 1,-1]+p[ 1,0]+p[ 1,1]
    if p and sum<2 or sum>3:
        p._ = 0
    elif not p and sum==3:
        p._ = 1
```

On a Pentium M 1.5GHz Windows laptop, NetLogo's implementation took 41 seconds to perform 50 updates and render operations and 35 seconds to do just the updating. In other words, the computations ran at .195 Msite/sec with rendering and .228 Msite/sec without rendering. The SIMP version ran in 2.5 seconds with rendering and .31 seconds without. The rates were 3.1 Msite/second and 25.5 Msite/second, respectively. In all, the SIMP versions were between 15 and 100 times faster than the NetLogo versions.

It's a bit unfair to compare SIMP, a special-purpose CA programming environment which generates C code on-the-fly, with NetLogo, which runs in pure Java code. NetLogo provides nice facilities for developing and rendering CA rules as well as a whole host of other parallel applications. In the future, it might be interesting to employ a STEP back-end to implement NetLogo-based CA programs.

8.1.2 Comparison with `C`

We found that we were able to code the Greenberg Hastings CA more than 5 times faster in SIMP than in C. The SIMP code even ran a bit faster than the C code.

As an experiment, we timed ourselves while we coded and validated the Greenberg-Hastings CA as a SIMP program and a C program. We finished the SIMP program in 14 minutes with 46 lines of code and about 5 debugging iterations. We finished the C program in 73 minutes with 68 lines of code and approximately 30 debugging iterations.

On `pm5.bu.edu` a 2.4 GHz P4 Gateway Linux desktop computer with the PcCodeGen engine the SIMP program ran 1024 steps on a 1024×1024 space in 12.3 sec or a rate of 87 Msite/sec while the C code did the same in 13.9 seconds or 76.86 Msite/sec. (The C code was compiled by GCC with full optimization turned on.).

In addition, the C program is certainly less flexible than the SIMP program. The C implementation requires boundaries sizes to be powers of 2. The SIMP implementation does not have this limitation. When validating the C implementation, we employed "printf" statements for rendering. When validating the SIMP implementation, we just used on-screen rendering.

8.1.3 Comparison with ZPL and related data-parallel languages

In this section, we compare and contrast the STEP architecture with ZPL, which we employ as a representative example of array languages. We choose ZPL because, out of a variety of array languages, it's data-parallel, region-based model comes closer to that of STEP and SIMP than most others. As such, it is a good representative for comparing and constasting our contributions in SIMP.

ZPL is a region-based, parallel array programming language (Chamberlain et al., 1998; Chamberlain et al., 1999b; Chamberlain et al., 1999a; Deitz, 2005). In its nature, it bears much resemblance to other array languages such as APL (Falkoff and Iverson, 1973). It also operates in a way that's similar to the parallel loop and array constructs available in High Performance Fortran (HPF) (Loveman, 1993) and Fortran 90/95 (Metcalf et al., 2004). The primary difference is that while these languages provide data parallelism with loop like operators and parallel array-wide operations, ZPL operates on array regions.

In contrast to these languages, aim of the STEP architecture and the SIMP programming environment is not to be a general-purpose array language, but, rather, to target CA applications first-and-foremost and to do so in a way that's natural and convenient for CA programmers and effeciently implementable in CA and LGA-like hardwares. The STEP architecture is more domain-specific than these more general array programming languages. In particular, the aim to make CA, LGA, and BPCA applications as simple to program as possible to describe and implement.

As discussed in Section 1.4 the STEP framework does not to perform arbitrary parallel computations. (For example, STEP does not break the crystalline uniformity of its logical architecture to handle global communication or concepts of parallelism such as nested parallelism(Blelloch, 2003) or multiassociativity (Herbordt and Weems, 1991).)

The Greenberg-Hastings CA in ZPL

For comparison purposes, we developed an implementation of the Greenberg-Hasgings CA in ZPL. As a non-expert ZPL programmer, it took roughly a fourty-five minutes to do this after reading the documentation and a few warm-up exercises. One might expect that that with more experience that the development time might reach of SIMP[1]. The code is relatively simple and we present it in-full below.

```
program Gh;
---------- Declarations ----------
config var
  n       : integer = 10;    -- problem size
  iters   : integer = 1000;  -- iterations

region
  R    = [0..n-1,   0..n-1  ]; -- problem region
```

[1]The SIMP program required approximately 15 minutes, but was also more full featured with rendering an interactive user-interface.

```
var
  C : [R] integer; -- state
  FIRE : [R] integer; -- temporary 'firing' value

---------- Entry Procedure ----------
procedure Gh();
var
   it : integer;
begin
  -- Initialize the state
  [R] C:=0;
  [2,n/2] C:=1;


  -- Iterate GH updates
  [R]
  for it := 1 to iters do
    FIRE:=C@^[-1,  0] | C@^[ 0, -1] | C@^[ 0, 1] | C@^[ 1,  0];
    if (C=0 & (FIRE=1)) then
         C:=1;


    elsif C=1 then
      C:=2;
    elsif C=2 then
      C:=0;
    end;
  end;
end;
```

Let's briefly discuss the code. Prefixes like '`[R]`' in a statement like '`[R] C:=0;`' indicate the region over which an operation applies. Regions are declared as ranges of index values and then employed to qualify the area of a set of arrays in which an operation is applied. The operator `@^` in expressions like `C@^[-1, 0]` references neighboring data of a site. The '^' is shorthand for a region that wraps around as a torus.

The ZPL code is relatively short. It only consists of 37 lines of code, which is better than SIMP's 39 lines of code. It also runs in a serial environment at a rate of 75.2 Msite/sec, performing 1000 updates of a 1024×1024 space 1024 times in a matter of 13.29 seconds on a 2.5 GHz Pentium 4 desktop machine. While it is not quite as good as SIMP's rate of 87 Msite/sec, it is quite close. (ZPL compiles its code to C. Like the PcCodeGen STEP under which these results were achieved, ZPL acts as a source-to-source code compiler.)

At this point, it would seem that the only advantage of using SIMP over ZPL is that, because SIMP is actually embedded in the Python programming language, one can leverage the Python syntax and libraries. ZPL, on the other hand, is mostly a standalone language. (In particular, with SIMP, one control the computations with Python. In ZPL, one must use ZPL to perform all flow control and program the logic completely within ZPL. ZPL does not provide built-in facilities for rendering. In SIMP one can use the built-in rendering that SIMP's renderer and console classes provide, or rely on external libraries such as PIL[2].) However, SIMP is more language-appropriate for representing CA. For example, because the regions are implicit—they refer to the entire CA space they need not be repeatedly specified in SIMP declarations. Most significantly, as we'll see momentarily when whe compare ZPL for implementing more complicated LGA constructs such as HPP's sublattices, STEP has a clear advantage in that it's easier for the programmer to express and lends itself to better machine-optimizations.

A parallel Fortran implementation of Greenberg-Hastings

The parallel Fortran implementations also make the core of the GH update relatively simple to express. One can express the parallel update loop with code like

[2]The Python Imaging Library (`http://www.pythonware.com/products/pil/`).

```
FORALL (I = 1:n-1,J = 1:n-1)
begin
    if (C(I,J)=(C(I-1, J) | C(I,J-1) | C(I,J+1) | C(I+1,J))) then
         C(I,J)=1;
    elsif C=1 then
         C(I,J)=2;
    elsif C=2 then
        C(I,J)=0;
    end;
end
```

The semantics of the `FORALL` loop are that right-hand expressions are evaluated before left-hand assignments are made. One thing that is more complicted in Fortran is that the indexes must be explicitly presented in the loop. In contrast, in SIMP and ZPL, they are implicit. Another thing that's more complicated is that one must manually arrange for the boundary wrap-around. (Alternately, one may use the `CSHIFT` operator to implement data movement first and then just interact each site with array operators. The `CSHIFT` operator only does a shift in one dimension, so one must break down a n-dimensional shift into n `CSHIFT` calls.)

Conceptually, there is more in the ZPL and Fortran models than what is needed for programming CA. This is not a shortcoming, but it does mean that the programmer must come to understand a more complicated model before beginning to program CA under ZPL. SIMP targets CA more specifically, so one might expect the learning curve to be shorter with SIMP. Another distinction is that the implementation model is simpler and thus allows one to implement a STEP back-end kernel for a given architecture or optimization strategy more readily than is possible with a broader language with more constructs that an implementation must instantiate. Further, the implementation kernel optimizations can be directed at the issues of performing an event update scan rather than more general operations.

An advantage of the SIMP/STEP architecture is that it supports a more "crystallographic" notion of data-parallel computing. In particular, as discussed in Section 4.5 and it provides a direct means of defining state-variables on arbitrary integer sublattices of an n-dimensional integer coordinate space. Array languages don't support this directly. The advantages of this direct support are that the programs more directly express intent and structure of the computation and that implementations can more readily optimize the implementation of the update scans.

To see this, consider, once again, the checkerboard lattice of HPP. Recall that, for this dynamics, as is the case for a number of LGA dynamics, the logical model is that of state variables that are allocated on a sublattice of the grid and are shifted to alternate cosets as time evolves—see Fig. 4·2 and Section 4.5.

A (mostly) straight-forward implementation of HPP in ZPL

For comparison we programmed HPP using ZPL. Our first cut was a simplified version of HPP where we allocated four state variables grid and then used a mask to specify the alternating cosets that are updated updated. Unlike the STEP implementations, we allocated state variables for every logical coordinate rather than just a subset of them.

```
var
  A : [R] boolean;
  B : [R] boolean;
  C : [R] boolean;
  D : [R] boolean;
  Even : [R] boolean;
```

and specify update event function that both shifts and interacts the data

```
procedure Event();
begin
   if ( (A@^[0,1]=C@^[0,-1]) & (B@^[-1,0]=D@^[1,0]) ) then
       A:= B@^[-1,0]; B:= A@^[0,1];
       C:= D@^[1,0];  D:= C@^[0,-1];
   else
       A:= A@^[0,1];  B:= B@^[-1,0];
       C:= C@^[0,-1]; D:= D@^[1,0];
   end;
end;
```

To implement the dynamics, the event was issued on two different cosets as specified by a computed boolean mask called `Even`

```
procedure HPP();
var
                  it : integer;
begin
[R]               Even:= (Index1+Index2)%2;

                  for it := 1 to iters/2 do;
[R with Even]        Event();
[R without Even]     Event();
                  end;
end;
```

The semantics of the HPP procedure given above are that the space is updated through alternating invocations on the even and odd lattice. First, sites in even coset, as indicated by the elements having value of ‘1’ in the constructed mask array `Even`, are updated. Then the sites in the in the other coset (which, for this lattice, happens to have a value of ‘0’ in the mask) are updated. This is variant of the red/black updating that happens to implement HPP. At alternating iterations, the present state of the data alternates between the cosets. Programming other sublattices that have more than two grid sites in their unit cell, such as that of polymer dynamics of Section 3.4 which has 8 sites in its unit cell, would require that one create more masks for the other cosets.

Coded in this way, we had 38 lines of code as opposed to 27 lines of code with the SIMP implementation. Because it's not possible to distribute data on an arbitrary sublattice under ZPL, it is necessary to allocate the state variables on the grid, rather than on a sublattice of it. Although the ZPL implementation achieved a respectable execution rate of 8.5 Msite/sec as compared with 19.5 Msite/sec under STEP.

We believe that the increased time was due mostly to the increased memory bandwidth as a result of the fact that the state-variable arrays are twice as large as they need to be. Another factor is that the implementation can not take advantage of the regular structure of the sublattice that's embedded in the `Even` mask used to specify it—the structure is represented in a mask, which is an arbitrary collection of bits with no notion of the structured patterns within it. The mask is another array that the implementation must read.

Another complication with this approach is accessing data. In order to acces data the current state of a variable, one would need to access data using an appropriate region, as in

```
procedure ReadEven(VAR);
begin
  [Rhalf] out :=  VAR#[Index1,(Index1%2)+Index2*2];
  return out;
end;
```

where `RHalf` is the region for the compact output array. It is defined as

```
region
  Rhalf = [0..(n-1)/2,0..(n-1)/2];
```

In this code, the `Index1` and `Index2` literals are ZPL builtin arrays that give the index values along the rows and columns of the arrays respectively. The '`#`' operator is an index operator. Notice that the expression '`(Index1%2)+Index2*2`' is used to generate the skewed indexes in the X dimension.[3]

This more complex way of accessing array values can be contrasted with a single SIMP expression, namely `VAR.array()`, that does the same thing.

Implementing HPP using packed array storage

For further comparison, simlar to Section 4.5.5 we also implemented a ZPL version of the program in which state variables were packed into an orthogonal array. This version of the program was a bit more complicated, requiring about 58 lines of code, but was necessary for getting more performance out of ZPL. In it, similar to the SIMP verion, we distinguish the shifts (LGA data propagation) from the collision event. As described in Section 4.5.5 the symmetry of the LGA is lost with this encoding and two different shifts are required.

[3] In Fortran 95 and High Performance Fortran (HPF) it is possible to program sublattice scans using '`FORALL`' loops with similar indexes. The problem with this approach is that the modulus and multiplication operators are unwieldy for the programmer and may add a significant overhead to the computation of array indexes.

```
procedure Shift1();
begin
       A := A_@^[0,1];
       B := B_@^[-1,1];
       C := C_@^[0,0];
       D := D_@^[1,-1];
end;

procedure Shift2();
begin
       A := A_@^[0,1];
       B := B_@^[-,1];
       C := C_@^[0,0];
       D := D_@^[1,-1];
end;
```

Note that we have also had to declare two different arrays, such as `A` and `A_`, for each state variable. One holds the present state and one holds the next-state. The update collision event is

```
procedure Event();
begin
   if ( (A=C) & (B=D) ) then
       A_:= B; B_:= A;
       C_:= D; D_:= C;
   else
       A_:= A; B_:= B;
       C_:= C; D_:= D;
   end;
end;
```

When issuing updates, must specify the two update phases

```
procedure HPP();
var
          it : integer;
begin
          for it := 1 to iters/2 do;
[R]           Event();[R] Shift1(); -- update phase 1
[R]           Event();[R] Shift2(); -- update phase 2
          end;
end;
```

Coded in this way, the ZPL program required 58 lines of code as compared to 29 lines of code for a similarly programmed SIMP application and 38 when it was programmed in ZPL using masks. However, this somewhat un-natural encoding was necessary in order to achieve maximum performance from ZPL. Indeed, the performance was improved by more than a factor of 4 with this encoding versus the previous, more natural implementation (38Msite/sec versus 8Msite/sec). While this performance is better than STEP's performance here, as discussed in the next section, we believe that we can bring the performance to this level.

Sublattices implemented with sparse arrays

It is possible in ZPL to use masks to specify sparse data allocations (Chamberlain and Snyder, 2001). (However, in our tests of this code, the software was unstable.) One could, in principle, employ a sparse-array representations in ZPL, or similar constructs offered by a language such as Matlab (Higham and Higham, 2005) in order to represent sublattice data.

In these sparse array representations, one typically defines a sparse mask that indicates where the array has values that differ from an implied value that automatically fills the rest of the array. While this notion can, by virtue of the fact that it can represent arbitrary subsets of the array, can be used to define sublattices, it does not maintain the logical structure of the lattice—the representation is a bitmap rather than a set of generator vectors. As a result of this, the implementation can not easily use the orthogonal array packing techniques that STEP employs. Instead, alternate, more complex data structures are needed to reprent the arrays. This would degrade the efficiency of access to these data structures in comparison to a packed array. Further, using masks and sparsity to represent sublattice structures burdens the user with keeping track of which array elements contain values of interest and which do not. This is handled automatically with STEP's sublattice constructs.

8.1.4 Comparison of three programming styles for HPP in SIMP

Programmed as a pure lattice gas, HPP required only 27 lines of SIMP code and 8 lines for defining a rendering rule. The rendering rule even preserves the local geometry of the HPP lattice gas. The CAM-8 version required 48 lines of code to pack the dynamics of HPP into a square grid while the old SIMP version required 37 lines of code to do the same. Neither of these preserved the geometry when rendering.

Table 8.1 compares three different styles of programming HPP: HPP as a LGA on a sublattice, HPP packed into an orthogonal lattice and HPP as a BPCA in the Margolus Neighborhood. Rate comparisons are on a 1000×1000 site space updated 1000 times using the PcCodeGen STEP.

implementation	computation rate	code size	comments
packed	19.5 Msite/sec	29 lines	skews geometry/rendering
sublattice	18 Msite/sec	27 lines	maintains geometry
BPCA	48.3 Msite/sec	33 lines	

Table 8.1: Comparison of various ways of programming HPP under SIMP.

HPP on the sublattice is the shortest and simplest version. Unlike the packed version it only needs one set of shifts (the packed version has two different sets of shifts).

The BPCA version is fastest on the PcCodeGen STEP. This is probably because the LGA case is slower due to cache contention as a result of the way PcCodeGen stores signals. Each signal is stored in its own independent array rather than having all state in one big array. the LGA case, the signals of a site are not stored near to each-other. Each is in a separate array. But, in the BPCA case they are. All the signals are in a single array (the different signals of the LGA version are stored in the four cosets of the block partitioning induced by the event's latice.)

In the BPCA, there is one state variable, which leads to a scan that indexes into two different arrays—one input array and one output array. In the LGA, there are four state variables, which the PcCodeGen step allocates as four different arrays. This leads to a scan that indexes into *eight* different arrays—four for the input signals, four for the output signals. (Recall that the STEP implementations double-buffer.)

To get better performance in the LGA case, one might consider a different signal allocation strategy—one that packs state variables into a single array. This hypothesis is validated by the fact that, in an older implementation that actually did store the state variables in a single array using a bit-packing strategy. At that time, the performance of the LGA was 41 Msite/sec on the same machine.

8.2 Performance of STEP Implementations

As discussed in Chapter 7, a number of factors affect the performance of a CA program. The two main factors are the number of neighbors and the size of the space being updated. Other factors include the complexity of the transition function and, if the transition function is implemented as a LUT, potentially the randomness of the inputs to the function.

For performing benchmarks we want a CA rule that generalizes directly to various numbers of neighbors and gives rise to 'noisy', chaotic sequence of states. The former property will allow us characterize the effect of adding more neighbors while the latter ensures that the computation is non-trivial and would fully exercise a lookup table. A dynamics that meets these criteria is is the PARITY CA. Parity adds its neighbors and sets the next state value to the sum of the neighbors modulo 2—that is, 1 if the sum is odd or 0 if the sum is even. Expressed with the XOR operator '^' the parity event on a binary signal `c` looks something like

```
def parity():
  c._ = c[0,0]^c[0,-1]^c[-1,0]^c[0,1]^c[1,0]^c[-1,-1]^....
```

8.2.1 Performance as a function of the size

First, we characterize the effect of the size of the space on the performance. To do this, we ran a 5-neighbor parity CA over spaces of various sizes and computed the sites-per-second.

Fig. 8·1 plots the performance on the PcCodeGen STEP of the two dimensional PARITY CA with 5 neighbors over variously sized square spaces. The sizes were increased in powers of 2. The performance data was taken on the machine `pm5.bu.edu`, a 2.4 GHz Pentium 4 desktop computer manufactured by Gateway and running Linux. The machine was quiescent at the time of the benchmarks and each data point represents the average of many iterations.

The shape of Fig. 8·1 is explained by the fact that, when the size of the space is small, API and boundary-handling overhead costs dominate. When the space is large, these overhead costs are amortized, but cache effects may come into play. We discuss each of these effects in turn.

When the size is small, the poor performance is due to the fact that there is an

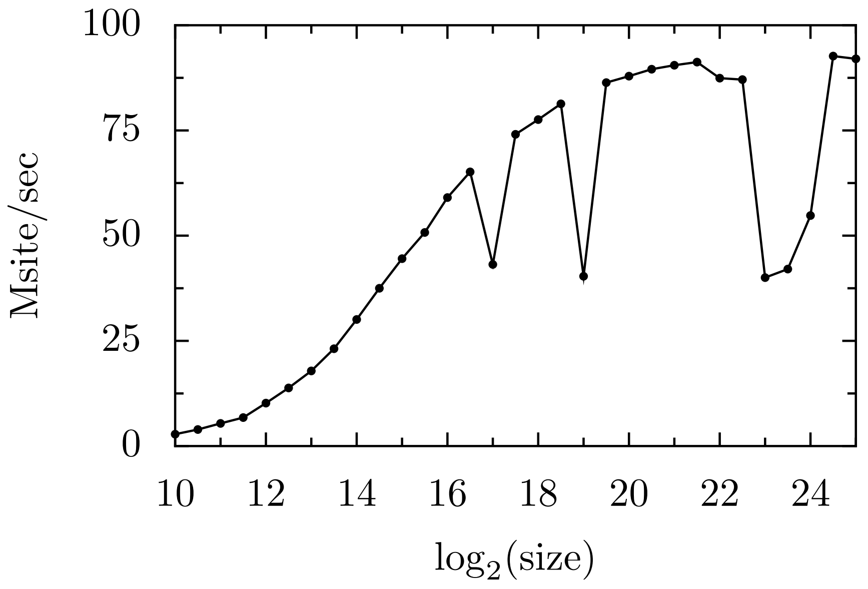


Figure 8·1: Performance of Parity CA running on PcCodeGen xversus size of the space

overhead of about 350 microseconds per event that we issue. We determined this by benchmarking against a modified version of PcCodeGen that does everything that the normal PcCodeGen does, except that it doesn't actually run the event loop's code.

Even if the 350 microsecond overhead were the only cost in running an event, in order to achieve performance of 40 Msite/sec, we would need at least 1400 sites to amortize the overhead alone. Not surprisingly, the effect of this overhead is visible for sizes up to $8192 = 2^{13}$.

After this, a different type of overhead dominates—the handling of boundary conditions as interrupts at the end of the scan lines. Increasing the size from 2^{13} where the lenght of a row is approximately 90 sites, up through 2^{20} where the size of a row is ten times larger at 1024 sites amortizes the boundary cost and improves performance.

Finally, after the size reaches 2^{18}, some particular sizes that have lower throughput than the others. Clearly some effect—we believe it to be the cache thrashing due to multiple scan lines competing for the same rows in cache—limits the computation to 40 Msite/sec. When this effect is not limiting the processing, there seems to be a compute-bound limit of around 90 Msite/sec. Because the point at which the degredation appears is around the sizes where a row in memory has 512 elements, we believe that cache thrashing is the likely cause.

8.2.2 Performance as a function of the number of neighbors

In general, we can expect that the performance will degrade as one increases the number of neighbors. There are two reasons. The first is that more neighbors mean that more data must be fetched—this costs in the number of instructions executed and increases the memory requirements of the working set. The second is that more neighbors mean that the transition function is more complex—if the transition functions is a LUT the complexity means that the LUT is larger and if it is implemented as a procedure, it means the procedure has more instructions.

Fig. 8·2 plots the performance of PARITY[4] running on three different STEP implementations—PcCodeGen,Pc, and PcCodeGen—as a function of the number of neighbors. We see that, indeed, the performance does decrease with the number of neighbors.

In this plot, we see that the PcCodeGen implementation is by far the fastest. This is because it resolves values in the scan loop to constants and the compiler can perform more optimization. The Reference implementation is so much slower than the even the Pc implementation that it's barely visable in the plot.

Implementing a transition function as a LUT versus as a C procedure has the

[4]The performance data was gathered under the same conditions as that of the previous size benchmark.

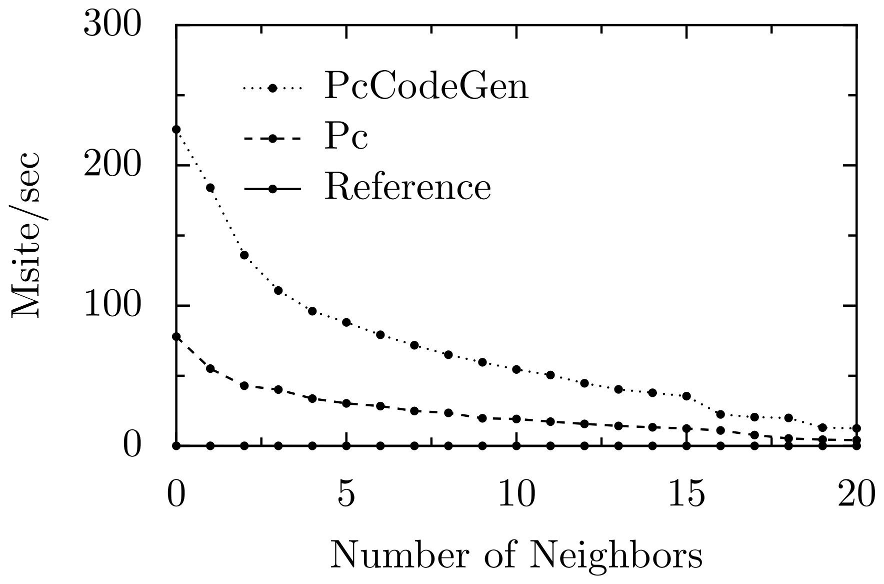


Figure 8·2: Performance of PARITY versus the number of neighbors

advantage that the LUT can essentially fold all the operations of the C procedure into a single table lookup. We take a moment to examine the tradeoffs.

The major disadvantage of using a LUT is that the size of the LUT is exponential in the number of bits of input. Other costs include the fact that one must precompute the outputs for all possible LUT inputs, and that, in order to perform a lookup, the inputs must be packed bitwise to form a LUT index.

The PARITY CA is an interesting case for comparing the performance of using a LUT to implement the transition function versus using a procedure. The PARITY CA has just about as simple of a transition function as is possible—it performs only one XOR per neighbor input. If, for a certain number of inputs, a LUT-based approach makes Parity faster, then, we can be fairly certain that it will for most other CA as well.

In Fig. 8·3, we compare the number of CPU cycles required to update a single site[5] in two different versions of the PcCodeGen implementation—one called "LUT" that implements the transition function as a LUT, and one called "C procedure" that implements the transition function with dynamically generated and compiled C procedural code.

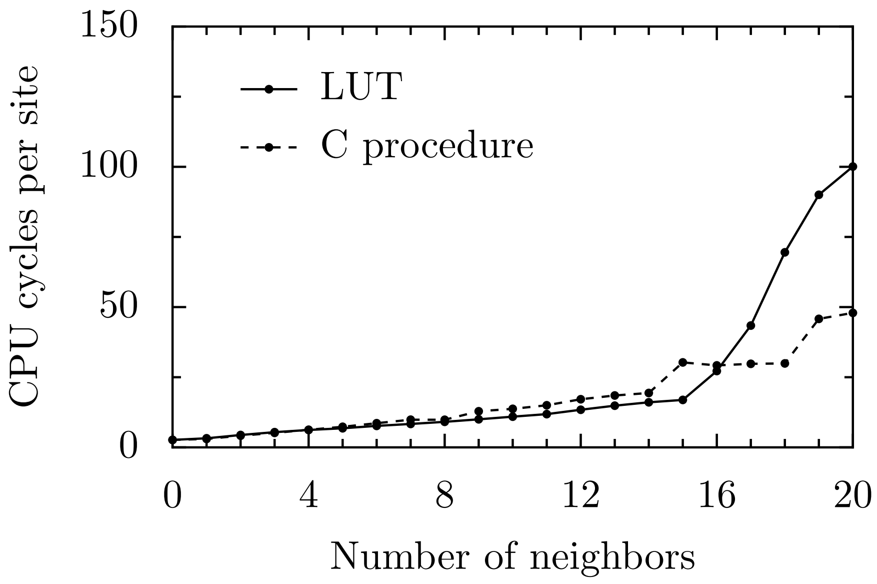


Figure 8·3: LUT and C-procedure performance versus number of neighbors

Although at times the speedup is slight, the LUT implementation requires fewer clock cycles per site and is faster than the procedural implementation up until the point at which there are 16 neighbor inputs/16 bits of input. After this point, the LUT approach becomes much more expensive and the procedural implementation is much more attractive. (Note that the value of 16 corresponds to 64K, which is the size of the L1 cache in the Pentium 4 processor.)

[5]PcCodeGen has a user-configurable parameter called `maxlutsize` which indicates the maximum number of bits that may appear in a transition function before it will switch from a LUT to a C procedure. To we set this to a large number for the "LUT" benchmarks and to -1 for the "C procedure" benchmarks.

[5]We use CPU cycles per site to measure the cost of updating a single site and compute by dividing the clock frequency of the CPU by the number of sites per second.

8.2.3 Parallelization

The PcThreaded STEP employs multiple threads to implement an Event by partitioning the space updated by the event into disjoint rectangular strips. Fig. 8·4 plots the performance of the 5-neighbor version of PARITY versus the number of threads that partition the space. The test was run on a 2000 × 2000 (4-million site) grid using the PcThreaded STEP running on a 32 processor system[6]. The plot depicts the performance up through 32 threads with a total of 32 processors utilized.

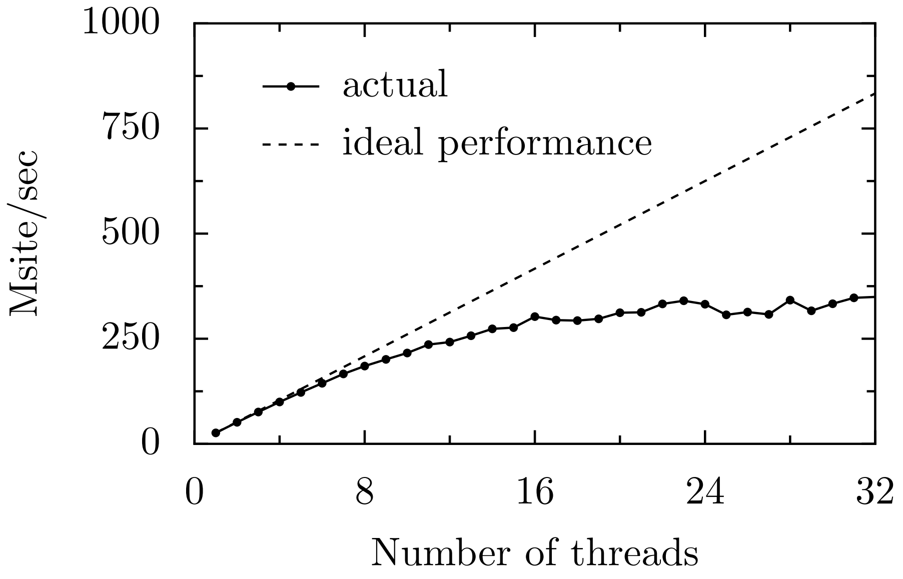


Figure 8·4: Parallel performance of PARITY5 versus number of threads

As shown in Fig. 8·4 and indicated by the data in Table 8.2, up through 8 threads the speedup is near ideal. Doubling the number of threads comes close to doubling the performance. However, after 16 threads, it seems that the shared memory communication costs begin to dominate any gain to be had by employing more processors. The absolute speedup levels off and the incremental speedup plummets to near zero.

[6]The system is `pobo.bu.edu`, a shared memory IBM pSeries 690 AIX 2.5 machine in the Boston University Scientific Computing and Visualization Center (`scv.bu.edu`). It has 32 Power4 processors, each running at 1.3 GHz and all sharing 1 GB of memory.

Doubling the number of threads again from 16 to 32 yields relatively little gain.

While it's likely that, with larger spaces, the speedup would further improve, this simple parallelization strategy is already sufficient for attaining a significant speedup on desktop CPU systems. Currently, such systems rarely exceed more than 8 processors. Clearly, though, more work is required before a larger number of processors can be utilized.

# threads	performance	absolute speedup	incremental speedup
1	25.6 Msite/sec	1.00	n/a
2	51.3 Msite/sec	2.00	2.00
4	100 Msite/sec	3.88	1.94
8	181 Msite/sec	7.18	1.85
16	302 Msite/sec	11.8	1.64
32	351 Msite/sec	13.7	1.16

Table 8.2: Parallel speedup PARITY5 versus number of threads

8.3 Cross Validation

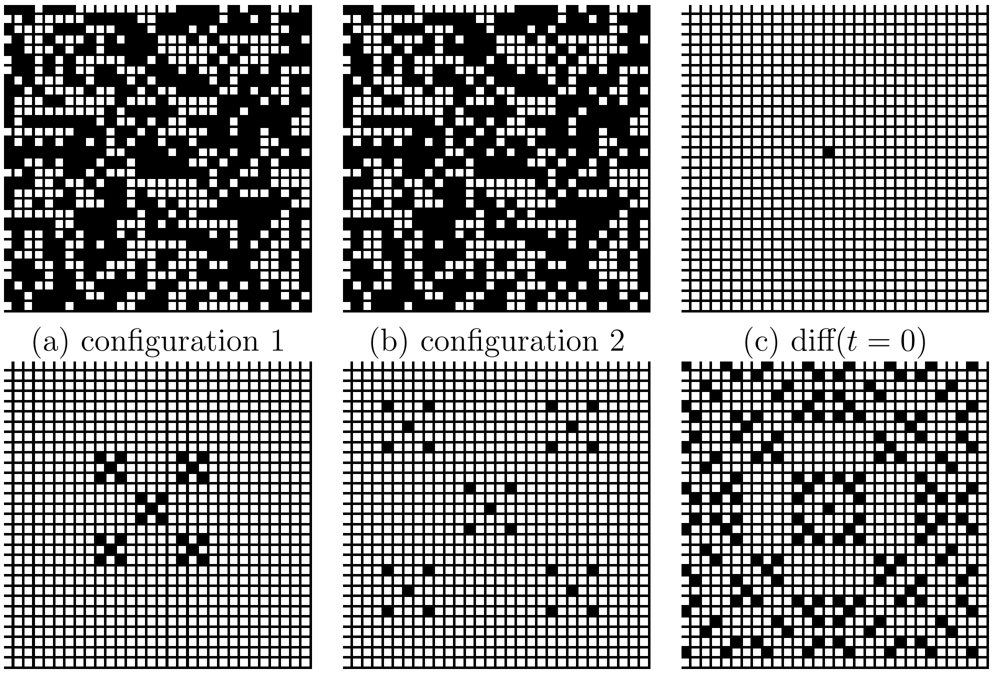
Introducing new constructs and capabilities to the STEP framework makes it more useful, but may tend to destabilize implementations, especially when they try to aggressively optimize the implementation of these constructs. To combat this, we cross-validate more advanced implementations against the Reference STEP. By virtue of being coded in a simple, reliable fashion the Reference STEP is a reliable 'yardstick' against which to quantitatively validate the correctness of an implementation.

The cross-validation strategy taken is simple. It runs through a set of CA programs implemented simultaneously on two different STEPs. The programs are initialized to the same conditions and employed to ,run the same dynamics. At various points, the results of the two are compared with one-another. If there is any difference, the cross-validation fails, otherwise it succeeds.

Because the Reference STEP is slow, we only use it to perform the first cross-validation, and having done this, perform others 'transitively' against the first STEP we validated. First the code cross-validates PC against Reference and then cross-validates all others against PC.

The types of CA programs we run to perform cross-validation are those in which information travels quickly and spreads globally. Because this is true of the parity CA, we use it to perform cross-validation as well.

In particular, consider two identical, but random configurations of binary signal that differs at just a single site. Fig. 8·5 (a) and (b) shows the bitmaps of two such configurations that differ in one bit. The bitmap in Fig. 8·5 (c) marks the initial bit difference in black. Because bits in parity propagate linearly, the bit difference propagates in a predictable fashion. Fig. 8·5 (d,e,f) show where the difference has spread after successive iterations if a von Neumann neighborhood parity CA.



(a) configuration 1 (b) configuration 2 (c) diff($t = 0$)

(d) diff($t = 5$) (e) diff($t = 10$) (f) diff($t = 15$)

Figure 8·5: Difference propagation in the parity CA

We employ the local parity transition function in one, two, and three dimensions. We also implement LGA and BPCA variants that employ related parity transition-functions. We also test the `GetCoset` and `SetCoset` operations. We implicitly test the `Read` and `Write` operations by employing them to set initial conditions.

Cross-validating the `Stir` operation is difficult because it's not uniform. One could, in principle, try to define some kind of statistical tests for this operator, but because it's usually so simple to implement, we don't bother to test it. Using these tests we quantatitatively verify the STEP implementations. The cross validation techniques are general and can run with any future STEP implementations.

Chapter 9

Conclusions

We have shown that, by analyzing the recurring constructs and concepts in the programming and implementation of CA, one can construct a architecture that affords the user a high level of abstraction and yet is amenable to efficient, flexible implementation. In this work we've motivated, proposed, implemented, and analyzed an architecture for computing with cellular automata called STEP. This framework includes a programming environment called SIMP, an abstract CA architecture called STEP, and an API that decouples the two.

We've demonstrated that SIMP can be used to program a variety of CA application types including ordinary cellular automata, lattice-gas automata and block partitioning cellular automata. To date, no other CA programming environment simultaneously supports all of these abstractions and this is a novel contribution of our work. To support it, we have added sublattice constructs for expressing and implementing user-level abstractions and constructs for programming lattice gas and block partitioning cellular automata (varieties of CA that mix concepts from the two). We have shown how these constructs can be represented in the STEP API and efficiently implemented on ordinary computers.

The result is that SIMP CA programs are qualitatively simpler and quantitatively smaller and faster to implement than low-level strategies like C and much faster than other high-level CA programming environments. Indeed, SIMP applications are smaller and faster to write than corresponding C programs. Although they are as short as other high-level scripting environments, their performance is equal to that of hand-coded C. And, because we've designed the STEP API in a way that allows one to implement a STEP in hardware, SIMP programs could potentially end up being be orders of magnitude faster than C.

Because the user would like to program at a high level in an environment with built-in, professionally designed general-purpose scripting and analysis facilities, SIMP is implemented as a Python programming language module. We demonstrate methods of extending the interpretation of pure Python code to handle constructs for advanced CA programming. This includes methods of automatically transforming Python specifications of transition functions into a low-level representation suitable for implementation in C or other optimized hardware or software.

Because the user wants CA programs to be fast and efficient, we've developed and presented techniques for efficiently implementing CA operations. In particular, we've developed a technique for efficiently handling boundary conditions in a way that does not adversely affect the speed of updating the larger volume. We've also implemented techniques for automatically implementing the transition function with a LUT or just-in-time compiled C code. All of this has lead to fast PC implementations.

The remainder of this chapter wraps up our thesis by noting the work of some STEP users, discussing the present state of and future of the STEP software, offering suggestions for further work, and, finally, some parting remarks.

9.1 SIMP users and collaborations

In addition to our own work with it, other colleagues and researchers have already employed SIMP to perform CA-based studies. With the recent addition of the STEP extensions for sub-lattices, BPCA, and enhanced types that support mixing binary, integer, and floating-point computations, we expect that this user-base will continue to grow.

Patrizia Mentrasti and Silvio Capobianco have used SIMP to study traffic models. Charles Turner has provided the project with some nice implementations from (Toffoli and Margolus, 1987). Frederic Gruau, Gabriel Moszkowski, P. Malbos, and Yufei Wang have also used SIMP to form a substrate for “Blob” computing (Gruau and Malbos, 2002; Gruau and Moszkowski, 2004; Gruau and Moszkowski, 2004). The collaboration with Gruau et. al has been particularly useful in driving the development process. In particular, this collaboration helped us to see the need for parameterizable event functions. We take a moment do describe their work.

In the ‘blob’ computing initiative Gruau et al. have employed SIMP to simulate cell-like blobs that act in CA spaces. Similar to von Neumann who employed CA as a substrate to create self-reproducing machines, Gruau employs CA to implement a dynamics that supports blobs. Fig. 9·1 depicts a ”blob” undergoing a fission process.

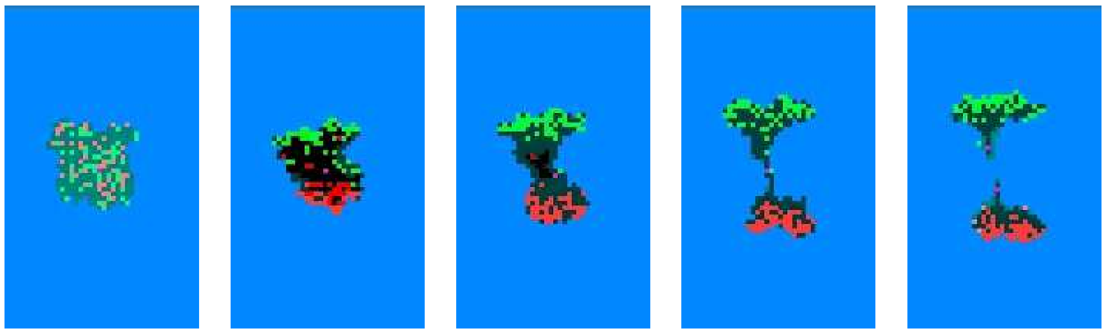


Figure 9·1: Blob fission

The idea of the blobs is that they will form a self-reproducing network of connected blobs capable of carrying out arbitrary computational tasks. At the implementation layer, Yufei Wang, in his Master’s thesis, (Wang, 2004), programmed a ‘blob’ as a kind of balloon based on the combination of two dynamics—the HPP lattice gas and a boundary model that, like the polymer model, represented the boundary with a chain of bits. The boundary was programmed to contract until the pressure from the gas it contained prevented it from contracting further.

The SIMP program that they originally wrote was programmed as a LGA and required many hundreds of lines of code. We estimate, based on our own programing of a similar dynamics, that the size of the program could be reduced to about 100 to 150 lines of code. The things that make this possible are 1) the constructs for programming with block partitioning cellular automata 2) the addition of parameterizable CA events functions and 3) the ability to declare multiple state variables.

9.2 About the software

Although we hope that our overall lasting contribution will be in our further refinement of an abstract architecture for cellular automata machines and techniques for programming them, the software instantiation of the STEP architecture is itself a concrete result. We take a moment to discuss some of the practical matters related to the software.

The software implementation of the architecture consists of the SIMP programming environment, the STEP API, and STEP implementations. It is the fruit of a substantial amount of work The software contains more than 15K lines of code including over 34 source files and a suite of applications. The software runs on all major platforms (Windows, Unix, Mac) and is distributed as source and binary installer format for a variety of platforms. Currently, the software is in version 0.7, reflecting the fact that it still has some work to undergo before a major stable release.

First, because we wish to encourage collaboration and the dissemination of knowledge, the STEP architecture is an open-source software project. In order to encourage collaboration, the software is released under the Library GPL (LGPL) license and made available through `sourceforge.net`, an internet-based collaborative "workshop" and rendezvous point for open-source software. The home page is `http://simpstep.sourceforge.net` and links to the software downloads appear there.

SIMP and STEP build on a variety of Python modules. As has already been mentioned, the STEP API extensively employs the `numarray` module, a Python multi-dimensional array interface. This array interface and the related libraries and utilities have made much of the software simpler to write and easier to extend. Further, because SIMP employs numarray objects for representing state variables at the user level, the SIMP programmer can automatically use the vast numarray libraries to manage state variable data. In particular, the numarray object includes facilities for saving, creating, analyzing, plotting and initializing numarray arrays.

The numarray software has recently gone through a number of re-designs and it seems that on-going work has moved to a new module called `numpy` (short for numeric python). Once this distribution has stabilized, we plan to replace SIMP's dependence on numarray with numpy. Numpy promises to be more efficient in the handling of small arrays and, in some future version, may actually be distributed with the base Python system.

Another important module on which the architecture builds is the `pygame` module. `pygame` is a convenient cross-platform, python-based interface to video buffers. The SIMP `Console` user-interface is built on the `pygame` (`http://pygame.org`) Python interface to the SDL (simple direct media layer).

9.3 Suggestions for future work

9.3.1 STEP API extensions

Should the demand arrive to create alternate bindings for STEP implementations, one might consider re-casting the STEP API in a lower-language like C or C++. The primary challenge in doing that would be in defining a common language for expressing transition functions. At present, that 'language' is a Python abstract syntax tree. Some other type of AST-like data structure would be needed.

Poisson events

One interesting extension might be a built-in operator for specifying events that use Poisson updating. The JCASim software actually does this. Although it is possible to program this by creating a shuffled state-variable that controls whether a local event has permission to execute (note that this is probably the simplest way for a STEP to implement it too), it may, nevertheless, be advantageous to have an operation for direct specifying a Poisson update event. This would help in the case where, say, a certain architecture has direct support for Poisson updating and would not naturally implement it with a stirred random variable. It would then be up to the architecture to figure out how to implement Poisson updating.

Alternate boundary types

There are a number of alternate boundary types one might consider. We did not focus on them in this work because they can, for the most part, be programmed within a dynamics. It's also unclear how the boundary types should be represented.

Here are few boundary types that might be worth implementing.

fixed Beyond the boundary, a signal has a fixed value. Interesting design aspects include how to represent these values.

reflecting Coordinates beyond the boundary, coordinates reflect back.

truncated Coordinates beyond the boundary stop at the boundary.

shear The grid is still on a torus, but rather than wrapping around modulo a rectangular lattice, they wrap around modulo some non-orthogonal lattice.

boundary-less The space logical size of the space is infinite. However, only a finite subset of the state-variables in the space have non-trivial values. This type of boundary is compatible with a sparse updating strategy where only the subset of the space with non-trivial, non-repeating state variable values is actually updated. The implementation would be responsible for determining, as the CA runs, which subset of the infinite space actually contains non-trivial values.

Other

Other mechanisms one might consider adding include

constant signal A constant-value signal. This be useful for some types of dynamics. Its values would need to be declared at initialization time.

transfer An operator for reading one signal region's values and writing them into another signal region.

randomize An internal operator for generating high-quality random states.

It might also be reasonable to add a "destroy" method to the STEP interface. Currently, a STEP can only be made to declare and instantiate signals and operations. It might be helpful if there were a way to destroy them. Destroying signals would be tricky, though. Destroying a signal would automatically invalidate any operation that depends on it having been defined.

9.3.2 SIMP programming environment

Much of the future work will be in adapting and tuning the SIMP programming environment. Later, it may be useful to tune the STEP API in order more easily/efficiently support recurring SIMP constructs.

The STEP programming environment is already quite flexible. By synthesizing the operations it provides, one can do things like

- declare generic CA events as Python functions and parameterize them at event declaration time,
- program interconnected CA subsystems that operate under a dynamics that's influenced by another dynamics—for example, one could employ HPP as a layer for generating "thermal noise" used by another system,
- mix state variables with different numbers of dimensions (for example, one can declare a one-dimensional state variable in a two-dimensional space),
- create *hierarchical* computations by making a "pyramid" of sublattices.

In what follows, we discuss some higher-level constructions that SIMP might enable including "programmable matter libraries" and 3D rendering.

Programmable matter libraries

SIMP is a good development platform; however, we would also like to develop libraries of re-usable CA sub-programs. Much work could be done in the construction of "programmable matter" libraries that allow one to select whole dynamics rather than just the low-level constructs used to implement them.

An example is the "tracer" material shown in Fig. 1·1. This dynamics is useful for tracing fluid flows in general, and need not necessarily be re-implemented every time that it's needed. Instead, it would be interesting to be able to just pass an argument giving state variables of the fluid to be traced and perhaps a parameter describing how the tracer material should be introduced into the flow. The same concept could be employed to make dynamics libraries.

In principle, STEP and SIMP already enable this to some degree. For example, one can define event transition functions as parameterized rules in some helper module and then bind values to the names used in these functions at the time that an event is declared. We have already used this technique, but have not really tested how well it could be used to create a generic library.

Another interesting library would be one for generating combined sets of random numbers using the strategy described by Smith in (Smith, 1994).

In all of these libraries, it may be the case that certain constructs will recur and that adding some SIMP helpers will make the libraries easier to write.

3D rendering support

An interesting, useful extension would be to include builtin tools for three-dimensional rendering. In doing this, one can make the choice of either performing rendering using external tools like OpenGl or doing the rendering in STEP.

Performing 3D volumetric rendering with OpenGl is probably the most general-purpose technique, but can be computationally expensive. As such, it may not be suitable for interactive rendering.

On the other hand, performing rendering within a STEP is reasonable. One can simulate 'light' as a dynamics in a STEP and employ it to obtain a 2D view of a 3D dynamics. Of course, though, having a full-blown 3D simulation of light is often more than what's needed and would incur more computation costs than necessary. In particular, in order for a "beam" of light to propagate through a n^3 space, it would take $\approx n^4$ site updates if the light dynamics is three-dimensional and propagates locally.

Actually, Fig. 1·2 (b) was generated using a *Z-buffering* like technique and only requires $\approx n^3$ updates. This simple rendering technique was also employed in the

CAM-8. It uses 2D signals in planar scans that march forward in the Z direction and develop a 2D image. For the figure, we employed 3 2-dimensional signals[1]—one for the output image and two for 'light' arrays. The rendering dynamics marches the three signals in the Z direction one step at a time, interacting them locally with a binary "material", rendering the value "0" as slightly opaque and '1' as a white solid. One of the light signals is shifted only in the Z direction while one is shifted at an angle in order to cast diagonal shadows that illuminate depth. If a light array encounters a solid site, it will add its value to the output image at that site, unless the output pixel has already been obstructed at the current depth. The event can tell if the view is obstructed by examining whether the Z light is present or has already been reflected. In order to implement opacity of the (mostly) clear material, we decrement the light value slightly at each step.

Although this strategy produced nice and efficiently generated results, it is tedious to reprogram a rendering dynamics for every different 3D simulation. It would be better to have generic libraries for performing rendering. On the other hand, a nice feature of programming the rendering in SIMP is that it's flexible. An interesting project would be to construct a generic STEP-based 3D renderer.

9.3.3 STEP implementations

We have only implemented a few of the possible STEP implementations. An obvious extension of this work is to implement other types of STEPs and compare their performance.

In Section 8.2 we have the beginnings of a benchmarking suite, but this could certainly be expanded. It would be good to expand the benchmarks to cover different types of applications including

[1]We did this by making the generator in the Z dimension as large as the space itself. SIMP has a shortcut for doing this—just make the generator vector zero.

- a sparse CA (one where most of the state is quiescent),
- integer and floating-point lattice-Boltzmann simulations.

Most modern CPUs include some single-instruction, multiple-data (SIMD) instructions. The most prominent example is the MMX extensions in the modern Intel architectures. In principle, using MMX instructions, it should be possible to update a CA's state in a way that's fast enough as to not be compute bound at all, but I/O bound. As such, it should be possible to improve the performance of the current STEP implementations by an order of magnitude, approaching the bus bandwidths of billions of bits per second.

The STEP architecture still needs to be tested against some hardware implementations. Because of its availability, the most promising and interesting candidate is a FPGA-based implementation. Doing so might require a further "single assignment" type of restriction to the description of the transition functions. Because they can be represented in a FPGA, it might also be advantageous to extend the API with more kinds of types (bit parameterizable floats, etc).

Another interesting thing would be to try porting to CAM8. This would rejuvenate an existing hardware, and would lead to a number of interesting problems and discoveries. The disadvantage is that the CAM-8 is currently only available in limited quantities, so the implementation would be more academic.

More work could be done in making memory access patterns more efficient. By partitioning the space into blocks, a STEP could make data more local in higher dimensions. This would require changing the scan interrupt routines—the routine would need to glue together disjoint blocks of memory. Nevertheless, this could be done efficiently. Moreover, a block-based approach could allow for more effective parallelization—the shared memory contention would be quite a bit smaller. It would also be an implementation that would lead to multi-computer implementations. A significant challenge would be in making the read and write operations efficient.

Implementing the transition function

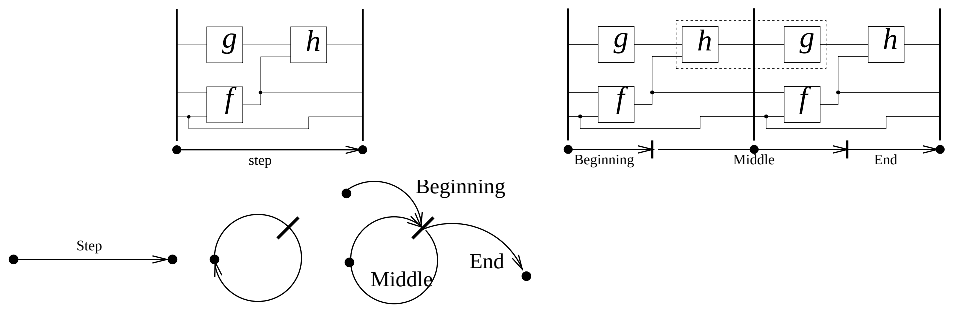
Many potentially advantageous transformations and optimizations require that the back-end have access to the substructure of the event transition function. We have provided a means—namely the AST, by which a STEP may access the local structure of a transition function and employ it to perform optimizations. Indeed, the PcCodeGen STEP already uses the AST to generate C code that implements the transition function.

However, there are a whole series of interesting implementation techniques that one might employ to efficiently implement the transition function in a given architecture. Motivational examples include the techniques employed in the CAMERON project which maps single-assignment C (SAC) programs onto FPGAs (Böhm et al., 2002; Rinker et al., 2001).

Below, we discuss a technique that's worth exploring—namely the possibility of implementing a complex transition function as a sequence of lookup tables.

Combinational networks of small LUTs can be mapped onto technologies supporting LUTs of greater or equal sizes through a process called LUT-packing(Cong and Ding, 1996). In general, LUT-packing attempts to find mappings that are minimal in terms of parameters such as delay, network complexity, and network depth. LUT-packing problems are related to the bin-packing problem and, consequently, are NP-hard. However, various suboptimal algorithms have been explored in the context of FPGA technology mapping (Cong and Hwang, 1995; He and Rose, 1994; Cong and Xu, 1998; Wilton, 2000). The LUT-packing and compilation problem addressed in STEP differs for various architectures such as the PC, in which a single, fixed LUT resource (the cache) is available at a given time. Hybrid architectures and FPGA's could employ multiple, parallel LUTs. In future work it would be interesting to explore some variants and identify optimization schemes for them.

As an example of LUT packing, consider the unrolled sequence of events in Fig. 9·2 ()b. The LUTs h and g can be combined into the single LUT marked by the dotted lines. This LUT straddles the original step boundary. If the back-end wants to use this optimization—especially across many steps—it has to break the step into a beginning segment to enter the loop, middle segment for the loop body, and an end segment to exit the loop. Fig. 9·2b and c depict the compilation of the loop into three segments—the circles mark the original step boundary, while the line marks the new loop boundary. This example closed and optimized the loop after one iteration, but optimizing over two, three or more iterations might also have been advantageous.



(a) a step (b) a reduction
(c) compilation into three segments

Figure 9·2: Function composition through LUT packing in the Temporal Loop

9.4 Parting remarks

The software instantiation of this architecture is already a practical result in itself. But we hope that our lasting contributions will be in our further refinement of an abstract architecture for cellular automata machines and techniques for programming them. The STEP architecture is an ongoing open-source software project. Section 9.2

discusses this project itself.

Indeed, we hope that the STEP architecture is more than software; it is *knowledge structure* for programming and implementing CA computing. In this thesis, we have hung a collection of concepts, theory, and techniques related to programming and implementing CA on this architecture. Operators like shifts and events and constructs like sublattices and cosets are about more than just programming CA—they represent ways of *thinking* about CA and CA applications.

In all, we hope that the STEP architecture will serve as a place where future innovators can hang work. CA programmers can write novel applications that, by virtue of their parsimony, are fairly easy to read and understand. Programming environment developers may find novel ways to synthesize other high-level constructs out of the basic STEP-API. Cellular automata machine designers can employ the STEP-API to shield themselves from application-layer issues and focus on fast implementations with the benefit of a built-in suite of applications to test against.

Our STEP architecture was not the first word in the area of CA programming environments and implementations and by no means do we expect it to be the last. Rather, our goal is to provide a working 'blueprint' for an integrated architecture approach towards programming and implementing CA. We have laid out a architecture for CA computing, drawn out the issues at play, constructing an exemplar architecture complete with a programming environment and implementations. The result is a successful proof of concept. We hope that this document and the accompanying software will show the way for the further development of the STEP architecture, or enable its central ideas to be incorporated in part or whole in some future architecture.

References

Bach, T. (2005). *The SIMP/STEP Reference Manual version 0.6.9.* Manual published on the internet `http://simpstep.sourceforge.net/`.

Bach, T. and Toffoli, T. (2003). SIMP/STEP: A rational framework for 'crystalline computing'. In *Proceedings of the World Multiconference on Systemics, Cybernetics and Informatics (SCI 2003)*, volume XIV, pages 312–320.

Bandini, S., Chopard, B., and Tomassini, M., editors (2002). *Cellular Automata: 5th International Conference on Cellular Automata for Research and Industry (ACRI 2002)*, volume 2493 of *Lecture Notes in Computer Science*. Springer.

Blelloch, G. E. (2003). Nesl: A nested data-parallel language (version 2.6). Technical report, Carnegie Mellon University School of Computer Science.

Böhm, W., Hammes, J., Draper, B., Chawathe, M., Ross, C., Rinker, R., and Najjar, W. (2002). Mapping a single assignment programming language to reconfigurable systems. *Supercomputing*, 21:117–130.

Chamberlain, B., Choi, S. E., Lewis, E. C., Lin, C., Snyder, L., and Weathersby, W. D. (1998). ZPL's WYSIWYG performance model. In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments.* (HIPS'98).

Chamberlain, B., Deitz, S., and Snyder, L. (1999a). Parallel language support for multigrid algorithms.

Chamberlain, B. L., Lewis, E. C., Lin, C., and Snyder, L. (1999b). Regions: an abstraction for expressing array computation. In *APL '99: Proceedings of the conference on APL '99 : On track to the 21st century*, pages 41–49, New York, NY, USA. ACM Press.

Chamberlain, B. L. and Snyder, L. (2001). Array language support for parallel sparse computation. In *Proceedings of the 15th ACM International Conference on Supercomputing.*

Cong, J. and Ding, Y. (1996). Combinational logic synthesis for lut based field programmable gate arrays. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 1(2):145–204.

Cong, J. and Hwang, Y.-Y. (1995). Simultaneous depth and area minimization in LUT-based FPGA mapping. In *FPGA*, pages 68–74.

Cong, J. and Xu, S. (1998). Technology mapping for FPGAs with embedded memory blocks. In *FPGA*, pages 179–188.

Cook, M., Rothemund, P. W., and Winfree, E. (2004). Self-assembled circuit patterns. *Lecture Notes in Computer Science*, 2943:91–107.

Culler, D. E. and Singh, J. P. (1999). *Parallel Computer Architecture.* Morgan Kaufmann.

Deitz, S. J. (2005). *High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations.* PhD thesis, University of Washington.

Falkoff, A. D. and Iverson, K. E. (1973). The design of apl. *IBM Journal of Research and Development*, 17(4).

Freiwald, U. and Weimar, J. R. (2002). The Java based cellular automata simulation system - JCASim. *Future Generation Computing Systems*, 18:995–1004.

Gosper, R. W. (1984). Exploiting regularities in large cellular spaces. *Physica D*, pages 75–80.

Greenberg, J. M. and Hastings, S. P. (1978). Spatial patterns for discrete models of diffusion in excitable media. *SIAM Journal of Applied Mathematics*, (34):515–523.

Griffeath, D. and Moore, C., editors (2003). *New Constructions in Cellular Automata.* Oxford University Press.

Gruau, F. and Malbos, P. (2002). The blob: A basic topological concept for hardware-free distributed computation. In Calude, C., Dinneen, M. J., and Peper, F., editors, *Unconventional Models of Computation, Third International Conference, UMC 2002, Kobe, Japan, October 15-19, 2002, Proceedings*, volume 2509 of *Lecture Notes in Computer Science*, pages 151–163. Springer.

Gruau, F. and Moszkowski, G. (2004). The blob division a "hardware-free", time efficient, self-reproduction on 2d cellular automaton. *Computing Frontiers*, 3141:317–337.

Hardy, J., Pazzis, O. D., and Pomeau, Y. (1976). Molecular dynamics of a classical lattice gas: Transport properties and time correlation functions. *Physical Review Letters A*, 13:1949–1961.

Harris, S. (2000). *Lattice-Gas Cellular Automata and Lattice Boltzmann Models : An Introduction.* Springer.

He, J. and Rose, J. (1994). Technology mapping for heterogeneous FPGAs. In *Proceedings of the ACM International Workshop on Field Programmable Gate Arrays.*

Hénon, M. (1987). Isometric collision rules for the four-dimensional FCHC lattice gas. *Complex Systems*, (1):475–494.

Hénon, M. (1989). *Discrete Kinetic Theory, Lattice Gas Dynamics and Foundations of Hydrodynamics*, chapter On the relation between lattice gases and cellular automata, pages 160–161. World Scientific.

Herbordt, M. C. and Weems, C. C. (1991). Multi-associativity: A framework for solving multiple non-uniform problem instances simultaneously on simd arrays. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume V, pages 219–223.

Higham, D. J. and Higham, N. J. (2005). *MATLAB Guide.* Society for Applied and Industrial Mathematics, second edition.

Hillis, W. D. (1986). *The connection machine.* MIT Press.

Ikenaga, T. and Ogura, T. (2000). Real-Time Morphology Processing Using Highly Parallel 2-D Cellular Automata CAM^2. *IEEE Transactions on Image Processing*, 9(12):61–71.

Ilachinski, A. (2001). *Cellular Automata: A Discrete Universe.* World Scientific.

Imre, A., Casaba, G., Ji, L., Orlov, A., Bernstein, G., and Porod, W. (2006). Majority logic gate for magnetic quantum-dot cellular automata. *Science*, 13:205–208.

Jacobs, L. and Rebbi, C. (1981). Multi-spin coding: A very efficient technique for Monte-Carlo simulations of spin systems. *Journal of Computational Physics*, 41:203–210.

Loveman, D. B. (1993). High performance fortran. *IEEE Parallel & Distributed Technology: Systems & Technology*, 1(1):25–42.

Mange, D., Stauffer, A., Petraglio, E., Peparaolo, L., and Tempesti, G. (2004). A macroscopic view of self-replication. In *Proceedings of the IEEE*, volume 92, pages 1929–1945.

Margenstern, M. (2006). A new way to implement cellular automata on the penta- and heptagrids. *Journal of Cellular Automata*, 1(1):1–24.

Margolus, N. (1994). *Pattern Formation and Lattice-Gas Automata*, chapter CAM-8: A Computer Architecture Based on Cellular Automata. American Mathematical Society.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344.

Metcalf, M., Reid, J. K., and Cohen, M. (2004). *Fortran 95/2003 explained.* Oxford University Press.

Resnick, M. (1994). *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds.* MIT Press.

Rinker, R., Carter, M., Patel, A., Chawathe, M., Ross, C., Hammes, J., Najjar, W., and Böhm, W. (2001). An automated process for compiling data flow graphs into reconfigurable hardware. In *IEEE Transactions on VLSI Systems*, volume 9, pages 130–139. IEEE.

Schulman, R., Lee, S., Papadakis, N., and Winfree, E. (2004). One dimensional boundaries for DNA tile self-assembly. *Lecture Notes in Computer Science*, 2943:108–125.

Segal, M. and Akeley, K. (1994). The design of the OpenGL graphics interface. White Paper.

Segal, M. and Akeley, K. (2001). The OpenGL graphics system: A specification (version 1.3). Technical report, Mountain View, CA,USA.

Smith, M. (1994). *Cellular Automata Methods in Mathematical Physics.* PhD thesis, Massachusetts Institute of Technology.

Snyder, L. (1986). Type architectures, shared memory, and the corrollary of modest potential. *Annual Review of Computer Science*, pages 289–317.

Spinellis, D. (2001). Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99.

Talia, D. (2000). Cellular processing tools for high-performance simulation. *IEEE Computer*, pages 44–52.

Toffoli, T. (1984). CAM: A high-performance cellular-automaton machine. *Physica D*, 10:195–204.

Toffoli, T. (1994). A fine-grained parallel supercomputer. Technical report, Philips Laboratory, Hanscom AFB, Department of Defense.

Toffoli, T. (1999). Programmable matter methods. *Future Generation Computer Systems*, 16(2):187–201.

Toffoli, T. and Margolus, N. (1987). *Cellular Automata Machines.* MIT Press.

Toffoli, T. and Margolus, N. (1990). Invertible cellular automata: A review. *Physica D*, 45:229–253.

U. Frisch, B. H. and Pomeau, Y. (1986). Lattice gas automata for the Navier-Stokes equation. *Physical Review Letters*, 56:1505–1508.

von Neumann, J. (1966). *Theory of Self-Reproducing Automata.* University of Illinois Press. Edited and completed by A. Burks.

Wang, Y. (2004). Blob displacement. Master's thesis, Paris South University & INRIA Futurs, France. supervised by Frederic Gruau.

Weimar, J. R. (1997). *Simulation with Cellular Automata.* Logos-Verlag.

Weimar, J. R. (2002). Cellular automata approaches to enzymatic reaction networks. In Bandini, S., Chopard, B., and Tomassini, M., editors, *Lecture Notes in Computer Science 2493 (Fifth International Conference on Cellular Automata for Research and Industry ACRI)*, pages 294–303. Springer-Verlag.

Weimar, J. R. (2003). Translations of cellular automata for efficient simulation. *Complex Sytems*, 14(2):175–199.

Weisstein, E. W. (2006a). Hermite normal form. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/HermiteNormalForm.html.

Weisstein, E. W. (2006b). Unimodular transformation. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/UnimodularTransformation.html.

Wilensky, U. (1999). NetLogo. `http://ccl.northwestern.edu/netlogo/`. Center for Connected Learning and Computer Based Modeling, Northwestern University. Evanston, IL.

Wilton, S. J. E. (2000). Heterogeneous technology mapping for area reduction in FPGAs with embedded memory arrays. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 19, pages 56–68. IEEE.

Winfree, E. and Bekbolatov, R. (2004). Proofreading tile sets: Error correction for algorithmic self-assembly. *Lecture Notes in Computer Science*, 2943:126–144. DNA Computers 9.

Wolfe, M. (1989). *Optimizing Supercompilers for Supercomputers.* MIT Press.

Wolfram, S. (2002). *A New Kind of Science.* Wolfram Media, Inc.

Worsch, T. (1996). Programming environments for cellular automata. In *Second Conference on CA and Industry.* ACRI.

Worsch, T. (1999). Simulation of Cellular Automata. *Future Generation Computer Systems*, 16(2):157–170.

Zuse, K. (1969). *Rechnender Raum (translated as Calculating Space).* MIT Project MAC. Tech.Transl. AZT-70-164-GEMIT (1970).

CURRICULUM VITAE

Ted Bach

Ted Bach was born in Athens, Ohio in 1976 to Jim Bach and Nancy Davis as Edward Alan Bach. He attended the Shade Elementary School in Shade Ohio and graduated from Alexander High School in 1994. He then took a year to save money and see the world, touring through Europe for four months on less than five thousand dollars.

Ted entered Ohio University in 1995 and in 1999 recieved the B. S. degree in Electrical Engineering with Computer Engineering Option. While at Ohio University, he served two terms as president of the IEEE and received a Leadership Award for the department of Electrical Engineering for his service. During these years, Ted also worked at Sunpower Inc. developing the user interface to Stirling engine simulation software and testing electronic circuits.

In 1999 Ted entered the Electrical and Computer Engineering department at Boston University and in 2002 received the degree of M. S. in Software Engineering. While at Boston University Ted studied cellular automata, pattern recognition, information theory, personal knowledge engineering, parallel computing, and software engineering. His primary research contributions were in several publications involving software support for cellular automata programming. In 2002 he consulted on information theory for US Genomics, a company working on ways of quickly detecting patterns in DNA.

Since 2005 Ted has been working full time for the Ab Initio Software Corporation of Lexington, Massachussetts helping companies to use Ab Initio Software to develop parallel applications for managing and transforming large volumes of data. At the same time, he has been finishing this thesis and starting a family. At present he has tired of writing about himself in the third person and is moving on to the next adventure.