Complex Systems **5** (1991) 63–88

Testing Parallel Simulators for Two-Dimensional Lattice-Gas Automata*

Richard Squier
Kenneth Steiglitz
Department of Computer Science, Princeton University, Princeton, New Jersey 08544

Abstract. We describe a test method for lattice-gas automata of the type introduced by Frisch, Hasslacher, and Pomeau. The test method consists of inserting test patterns into the initial state of the automaton and using a graphics display to detect errors. The test patterns are carefully constructed limit cycles that are disrupted by errors occurring at any level of the simulator system. The patterns can be run independently to test the system for debugging purposes, or they can be run as sub-simulations embedded in a larger lattice-gas simulation to detect faults at runtime. We describe the use of this method on a prototype parallel machine for lattice-gas simulations, and discuss the range of systems that can make use of this type of test method. The test patterns detect all significant one-bit errors. We include experimental results that indicate multiple-bit errors are unlikely to escape detection.

1. Introduction

Since Frisch, Hasslacher, and Pomeau [1, 2] introduced the use of lattice-gas automata to simulate hydrodynamics, their FHP models [3] and variants have been used in many simulation studies. Some simulations have used commercial supercomputers or parallel processors (see references [4–13], for example) and others have used special-purpose hardware [14–16]. Because it is not yet known how well lattice-gas automata model physical systems, there has been interest in comparing lattice-gas simulations with theoretical and experimental results. The validity of such comparisons depends on the correctness of the implementation. (The situation is illustrated in figure 1.) It is not usually possible to establish independently the correctness of an implementation because of the complexity of the operations used in the implementation. For instance, the complexity of floating-point arithmetic

*A condensed version of this paper was delivered at the International Conference on Application Specific Array Processors, Princeton, New Jersey, September 1990.

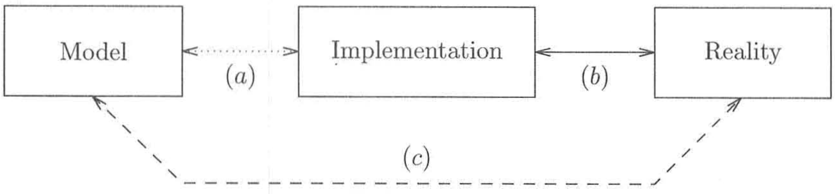

Figure 1: The solid line (b) shows the comparison we really make when we compare a computer model of a system to experimental results for that system. We would like to say that the comparison we are making is effectively between the model and the physical experiment (dashed line (c)). The dotted line (a) suggests the missing piece of information that would allow us to say this with conviction: the knowledge of the correctness of the implementation.

makes verifying the correctness of an implementation of a finite-difference scheme for integrating the Navier-Stokes equations impossible in practice. The state of affairs for lattice-gas simulations is quite different: the data movement and logic operations are simple. It is therefore possible and practical for the correctness of an implementation of a lattice-gas automaton to be tested exhaustively before runtime and monitored during runtime.

We became interested in verifying the functional correctness of a lattice-gas simulator while running fluid flow simulations on a custom VLSI processor, LGM-1 [16], built here as an experimental prototype. The particular simulation project we undertook involved comparing our simulation results for a specific flow problem with the results from other methods for the same problem. In making these comparisons we discovered that it was impossible to determine whether the discrepancies we saw were caused by the differences between the methods or by artifacts of incorrect implementation of our system. Furthermore, interspersed with simulations we were also modifying both the hardware and software of the system, requiring a concrete testing method for debugging purposes. From this experience we realized that a complete system testing method was needed that could be run independently of any simulations to verify functional correctness of the simulator system.

Our experience with simulations on LGM-1 also convinced us that system functional testing was not sufficient: we also needed runtime fault detection. We often ran simulations continuously for 24, 36, and more hours, and discovered that, aside from the errors caused by incorrect implementation of the algorithm, there were other sources of error of a more transient nature. For instance, we found that during long runs the host system or the network facility could cause errors, even though the system did not crash and the simulation ran to completion. Similarly, temporary failure of custom chips, pin connections, and so forth, could occur during a simulation, and not be detectable either before the run or after its completion. Although we realized that detecting every transient error during simulation was probably not possible, we guessed that the most likely kind of transient error was not the random single-bit error, but failures that would affect large pieces of the simulation, large either in time or space. We therefore began to look for ways of embedding runtime fault detection in the initial state of the simulated automaton.

In this paper we describe a testing method for lattice-gas simulators. The method can be categorized as specification-based system-level functional testing [17] because we use the specification of the behavior of a lattice gas to derive input data that tests the correct functional operation of a simulator system. This means we do not do any modeling of hardware faults although the test method is used to test custom VLSI chips of a special-purpose hardware simulator. Our approach consists of using graphic display of the lattice-gas state to detect errors in the evolution of cyclic sub-lattices. The collection of sub-lattices exhaustively exercises the update logic of the simulator, and is built from a small library of hand-coded test pattern templates. The test system consists of the template library, a small function library for creating patterns in a lattice-gas state, and a library of routines for image manipulation and display. In practice, we have used the test method in a "prevention oriented" manner [18] in the construction of the testing facilities themselves. That is, during construction of the template library, image manipulation, display software, and a simulator for the special-purpose hardware, the test patterns were used to prevent, detect, and debug implementation errors. Using a "destruction oriented" approach we have used the test method to test custom VLSI chips, custom circuit boards, and general control software of the special-purpose system. Finally, we have used the testing facility to embed test patterns in the input data of lattice-gas simulations used in fluid flow experiments to indicate simulator system functional runtime errors.

The remainder of the paper is organized as follows. Section 2 briefly describes the FHP-III and LGM-1 lattice-gas models. Section 3 introduces some terminology and definitions. Section 4 gives a general description of the test ensemble and some description of the methods used in its construction. Section 5 gives a detailed description of a single test pattern template. Section 6 describes the construction of specific collections of test patterns that constitute the test ensemble. Section 7 discusses the error detection and experimental results for multiple-error coverage. Section 8 discusses the issue of applying the test method to different architectures. Section 9 contains a summary of our experiences using the test method, and our conclusions.

2. Lattice-gas automata

Our version of a lattice gas is based on the FHP-III model. As described in [3] this type of lattice-gas automaton consists of a two-dimensional lattice graph [19] and a set of update rules for variables associated with each node in the lattice graph. The lattice is the triangular lattice on the plane generated by the unit vectors $\mathbf{e}_1 = (1, 0)$ and $\mathbf{e}_2 = (1, \pi/3)$, in polar coordinates. The edges of the graph connect nearest neighbors in the lattice (see figure 2). In cellular-automata terms [20, 21] each site together with its variables constitutes a

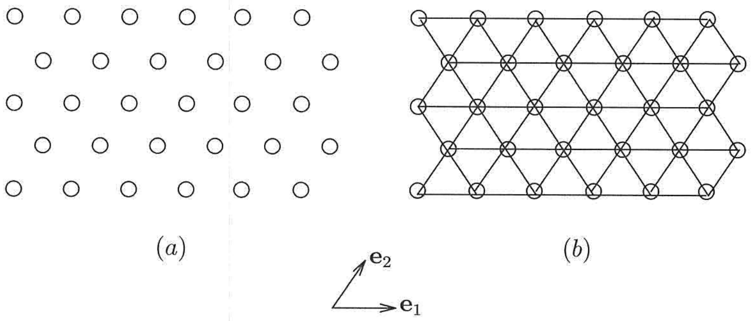

Figure 2: (*a*) A finite lattice generated by the unit vectors $\mathbf{e}_1$ and $\mathbf{e}_2$. (*b*) The lattice-graph produced from (*a*) by nearest neighbor connections.

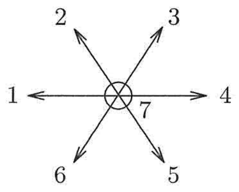

Figure 3: The seven velocity vectors at a lattice site. The seventh is represented by a circle and has zero magnitude.

cell of the automaton; the edges define the cell's neighborhood. Each cell has eight bits of state information: seven one-bit dynamical variables and one bit defining the type of site. In the lattice-gas view, at each site each incident edge has an associated variable representing the presence or absence of a unit mass particle with unit velocity directed toward the site's neighbor along that edge (see figure 3). The seventh dynamical variable encodes the presence or absence of a unit mass particle with zero velocity positioned at the lattice site, called a *rest particle.* The eighth bit encodes the presence of a barrier at the lattice site. With this interpretation of the variables the update rule is designed so that, letting the edges have unit length, the lattice is populated with particles traveling along graph edges and colliding at lattice sites (see figure 4).

A lattice-gas automaton evolves by synchronously updating the state of every cell of the automaton: the next state of a cell is determined by the states of its neighbors and its own state, and the automaton's update rule table. For the lattice gases the automaton update rule table is called a *collision rule set*, the initial configuration of states determining a cell's new state is called a *collision*, the next state entry in the rule table is called the *result* of the collision, and the combination of a collision and its result is called a *collision rule.* A particular lattice gas is defined by specifying a collision rule set that gives the results of every possible collision. These rules can

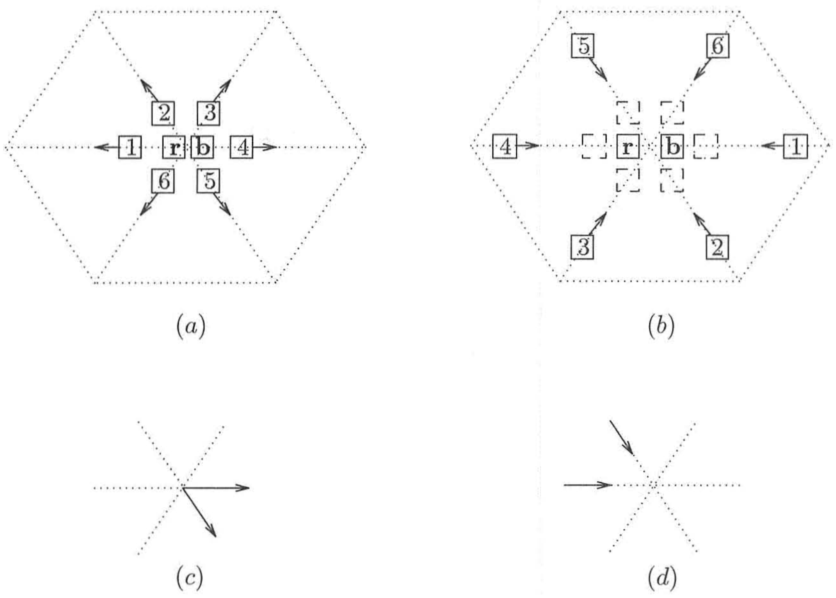

Figure 4: (*a*) The state of a lattice site and associated one-bit variables. **r** represents a rest particle, **b** a barrier site. (*b*) The input variables that affect the next state of the lattice site. (*c*) Shorthand notation for the state of a lattice site. (*d*) Shorthand notation for the input to the next-state computation. Input state (*d*) followed by site state (*c*) represents an update rule: (*c*) is the next state of the lattice site after an input of (*d*).

be thought of as rules about the action of particles (variables set to one) or equivalently as rules about the action of holes (variables set to zero), as they collide at lattice sites. In an FHP gas, particle collisions generally conserve momentum and mass, and are symmetric with respect to rotation by integer multiples of $\pi/3$, and time reversal; and for the FHP-III gas, collisions are also symmetric with respect to hole/particle duality (complementation of the dynamical variables).

These symmetry properties and hole/particle duality make it possible to define a collision rule set in a compact way: for each equivalence class of collisions induced by equivalence under duality and rotation transformations, pick one example, called a *canonical collision*, and show its next-state result. The combination of canonical collision and result is called a *canonical collision rule.* There are 28 canonical collisions for the eight-bit FHP-III and LGM-1 lattice gases, and figure 5 shows the fourteen "non-barrier" canonical collision rules for these two automata. Using the list of fourteen canonical collisions, c_0 through c_{13}, all 256 possible collisions for the LGM or FHP-III

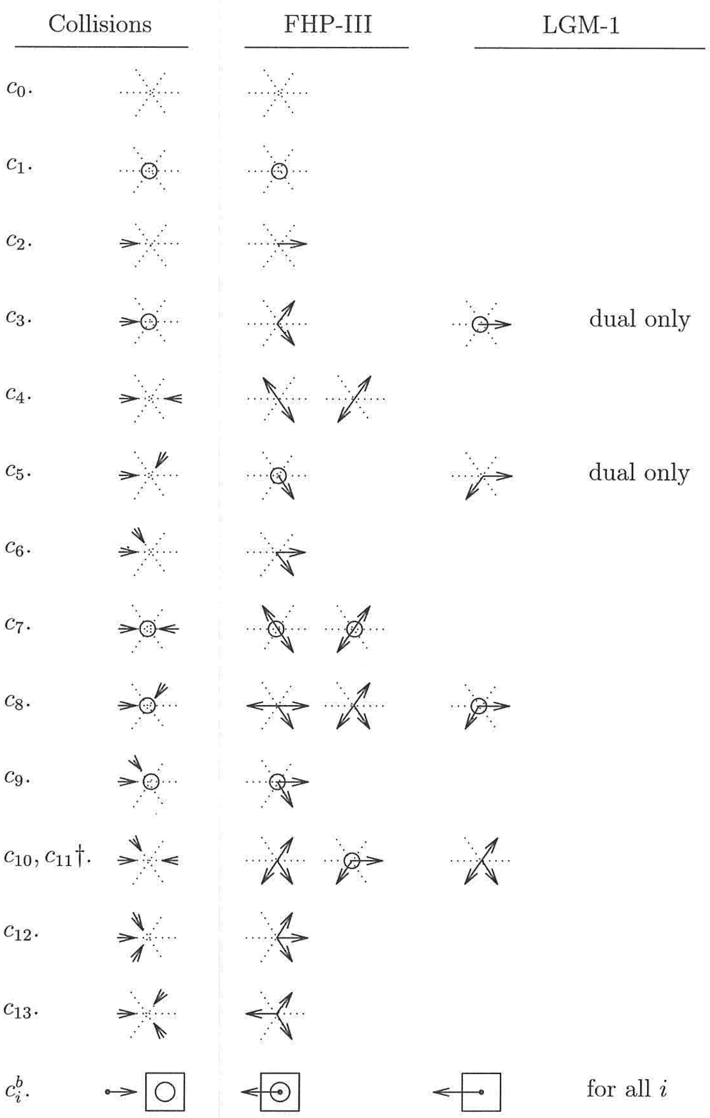

Figure 5: The rule sets for the FHP-III and LGM-1 gases. The first column shows all canonical collisions, c_0 through c_{13}, and a single example of the barrier version, c_i^b. The second column shows the results for the FHP-III gas. The LGM-1 rules are shown only in the cases where they differ from the FHP-III gas. (†) The canonical collisions c_{10} and c_{11} are reflections of each other through a horizontal line. Shown is the canonical rule for c_{10}.

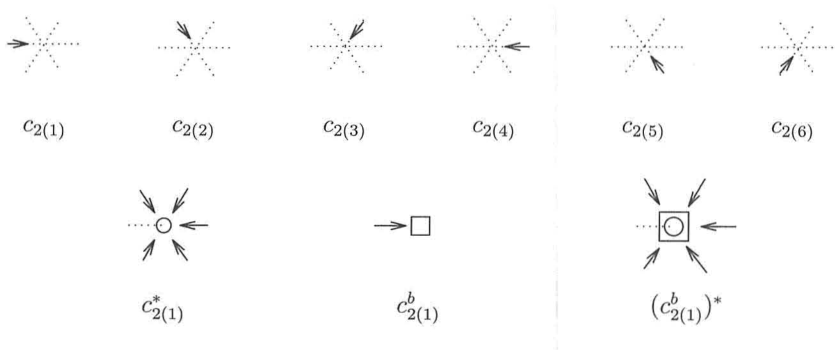

Figure 6: The actual collisions generated by canonical collision c_2. There are 24 total: six rotations for each of c_2, c^*_2, c^b_2, and $(c^b_2)^*$.

gases can be generated by applying the symmetry transformations plus barrier presence or absence. When the barriers implement the "non-slip boundary condition," the remaining 14 rules can be stated in a single sentence: all particles colliding at a barrier site reverse their direction of travel, and the last row of figure 5 shows a generic example.

The second column of figure 5 shows the results of the canonical collisions for the FHP-III gas. Where there are two result entries for a single canonical collision, we mean that the two possible results occur equally often. For instance, figure 5 shows two possible results for the c_4 collision, which LGM-1 implements by using one result on even rows and the other result on odd rows.

The third column of figure 5 defines the LGM-1 gas. Only the results that differ from FHP-III are shown. The c_3 and c_5 results are different for the dual cases only, while the c_8, c_{10}, and c_{11} results differ for both non-dual and dual cases. Because of the lack of dual symmetry in the c_3 and c_5 collisions, we have been mildly deceptive in figure 5: the c_3 and c_5 collision classes are not equivalence classes for the LGM-1 gas. Nevertheless, we shall continue to speak of them as if they were, pointing out the difference when necessary.

There is one other difference in the LGM-1 rules: a rest particle at a barrier site vanishes. This violates conservation of mass, but does not change the behavior of the lattice gas. Because this last property of the LGM-1 gas creates some difficulties in testing, we will have more to say about this when we discuss the construction of a complete test set for LGM-1.

The canonical collisions can be used to specify a particular collision by providing the rotation, duality, and barrier status. The collisions listed in figure 5 are presumed in rotation 1, written $c_{k(1)}$ for collision c_k. Clockwise rotation by increments of $\pi/3$ are written $c_{k(2)}, c_{k(3)}, \ldots, c_{k(6)}$. The dual of a collision is written c^*_k, and the presence of a barrier is written c^b_k. For example, figure 6 shows that canonical collision c_2 generates 24 distinct collisions: six rotations, their duals, six rotations with barriers, and their duals.

3. Some definitions and notation

Before beginning a discussion of the test mechanisms, we need to establish some definitions and conventions. As we described above, a particular lattice-gas automaton consists of a rule set, a lattice-graph, and the variables associated with the elements of the lattice-graph. Throughout the following we will assume the rule set and the form of the lattice-graph are both fixed and correspond to the LGM-1 lattice gas unless otherwise stated. Consequently, we may think of any such automaton as a system of individual particles that obey the collision rules and exist on a lattice-graph. An instance of a lattice-graph together with a specification of the barrier sites we call a *space*, a combination of a collection of particles and a space we call a *system*, and an arrangement of particles in the space we call a *state* of the system. With each system there is an associated time t defined by assuming some specified initial arrangement of a system assigned time $t = 0$, and each subsequent update of the automaton state increases t by one. A subsystem may have a local time t_{local} with an origin different from the global time origin. We write local time as a function of global time: $t_{\text{local}}(t)$. The idea is that an initial state of a system is built up from smaller systems whose states at global time $t = 0$ are achieved by starting the local systems in their initial states and running them forward or backward to $t_{\text{local}}(0)$. Thus, a local system can have its time origin shifted from the global clock's time origin. A system is termed *cyclic* if in its evolution it ever repeats a state. A system is said to be *closed* if boundaries are placed in such a way that particles outside the system can never enter. When we wish to indicate the state of a system Γ at time t we write $\Gamma^{(t)}$. For the period of Γ we write δ_Γ.

A collision is said to *occur at time t* in system Γ if the collision exists at some site in $\Gamma^{(t)}$. If a collision occurs at some positive t in Γ we say Γ *contains* the collision. When we discuss systems that detect evolution errors we will want to know not only if a collision is contained in the system, but also whether the system detects bit errors in the result. We will say a system *covers* a bit error if the system detects the error. There are eight possible one-bit errors in any collision result, which we denote $\{e_r, e_b, e_1, e_2, e_3, e_4, e_5, e_6\}$, corresponding to the eight variables associated with a lattice site. When we wish to specify a particular error in a particular collision we will append the bit-error notation to the collision notation with a colon separating the two. For instance, $c_{2(1)} : e_r$ shows that the result of a c_2 collision at rotation 1 has the rest particle bit set to the complement of its value in the correct result.

4. Test sets and their constituents

A particle system that tests a simulator for all one-bit errors in the evolution of a lattice gas we call a *complete test ensemble*. A test ensemble consists of a collection of cyclic subsystems called *test cycles*. Test cycles are built from elementary generic subsystems called *pattern templates*. (A *pattern* is produced from a pattern template by setting parameters related to the size of the pattern, its spatial orientation, the number of particles contained, and so on. Any such pattern could serve as a template, and we shall sometimes refer to templates and patterns interchangeably as patterns.) A test cycle consists of a collection of subsystems derived from a set of pattern templates by applying symmetry operations and time translations to the patterns produced from the templates. A test cycle can constitute a portion of a complete ensemble, or it may be inserted into a larger system simulating a fluid flow problem, to serve as a runtime error detector.

While one might consider trying to compile a pattern template set that has as few members as possible—each one covering as many collisions as possible—we found that a simple greedy algorithm led to a set of templates that let us construct a complete test ensemble that was small enough to be practical. To be more specific we used the following procedure:

1. Select a canonical collision.
2. Design a pattern template that contains that collision.
3. Simulate every one-bit error for that collision and check to see that the error is detected by that pattern. If a single-bit error is not detected, return to step 2 for a new test pattern template. Record which canonical collisions are covered.
4. Repeat steps 1 through 3 until all collisions are covered.

Note that it is only necessary to simulate bit errors (step 3) for an update of the canonical collision. If an error is detected in the canonical configuration, every bit error in its equivalence class will also be detected by an appropriately transformed version of the pattern.

After we had a complete set of pattern templates we could proceed with the construction of a complete test ensemble. Starting with an empty ensemble and using the test pattern collision coverage table, we continued the procedure:

5. Select a canonical collision/one-bit error combination not yet covered by our ensemble of test cycles.
6. Select a pattern covering that error.
7. Build a test cycle from the pattern by applying all the symmetry transformations to the pattern so that all actual collisions generated by the canonical collision are contained in the test cycle.
8. Complete the test cycle started in step 7 by making time-delayed versions of the system constructed in step 7. (We will explain this in detail later. The reason for time-delayed versions is to ensure that every processor in a pipeline will "see" the collision.)
9. Continue in this way, checking off the covered collisions until all one-bit errors are covered by our ensemble of test cycles.

Our complete set of test pattern templates contains 51 templates. Each pattern is enclosed in a square 30×30 box of barriers. If we naïvely build the test cycles for the LGM-1 lattice gas, we need, for each pattern, six rotations of the pattern and its dual, and about three different time-delayed versions of each of these. In addition, a complete ensemble for the LGM-1 architecture requires every collision on both even and odd rows. Altogether the complete ensemble built this way requires close to 3500 copies of the patterns. On the host machine for LGM-1, a SUN 3/160C workstation,[1] we can display about 1000 patterns per screen, and thus we can see the entire ensemble in four screens of bitmapped graphics. In practice, we reduce this considerably in two ways. First, we skip step 8 above: no time-delayed versions are required if we make the pipeline length relatively prime to the least common multiple of pattern cycle times. The length of our pipeline has been set to a prime number of stages. Second, we are slightly more careful in constructing the ensemble: not every pattern requires all twelve combinations of rotations and duals. The ensemble we use for a complete system test contains 786 patterns and its display occupies about three-quarters of the host's screen. We will return to the topic of display when we discuss error detection, but now we want to describe patterns and test cycles in detail.

5. A test pattern

Patterns are generic, self-contained subsystems designed to contain one or more canonical collisions. Our collection of pattern templates is divided into groups of similar type designated by letters A through K. There may be more than one version of a particular type; for instance, type K has twenty-eight versions K_1 through K_{28} (see [22] for a catalog of the complete template library).

The patterns are built using a small library of particle/barrier devices, which act as control mechanisms for collections of moving particles and individual moving particles. Groups of moving particles are called *chains*. A chain is simply a collection of particles laid out uniformly in a line, all following an identical path. The control mechanisms for chains consist of devices to deflect a chain, called *turns*. There are also mechanisms called *gates* that reflect single particles at only one phase of a cycle while letting particles pass at all other phases, and *detectors* that detect the incorrect presence or absence of a particle involved in some cyclic collision. We will discuss a few of the components in detail below as we describe the construction of pattern A. Descriptions of the other components can be found in [22].

Pattern A was designed to test the c_3 and c_5 collisions, and consists of three chains traveling around a roughly triangular circuit (see figure 7). The chains are the "paired" type: they consist of pairs of particles such that the members of a pair are separated by an empty site. The turns in A are designated type T_1 turns, and are designed to handle paired chains (see figure 8). The collisions contained in A occur, for the most part, in the turns,

[1]SUN is a trademark of Sun Microsystems, Incorporated.

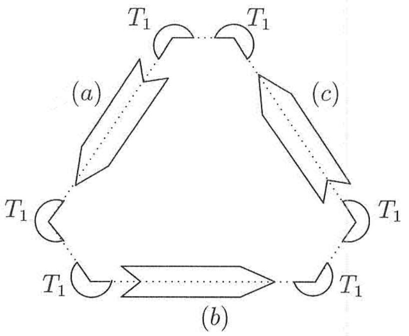

Figure 7: Pattern A schematic. (a), (b), and (c) are chains of particles with velocities $\mathbf{v}_6$, $\mathbf{v}_4$, and $\mathbf{v}_2$, respectively. The deflectors at the corners are type T_1 turns.

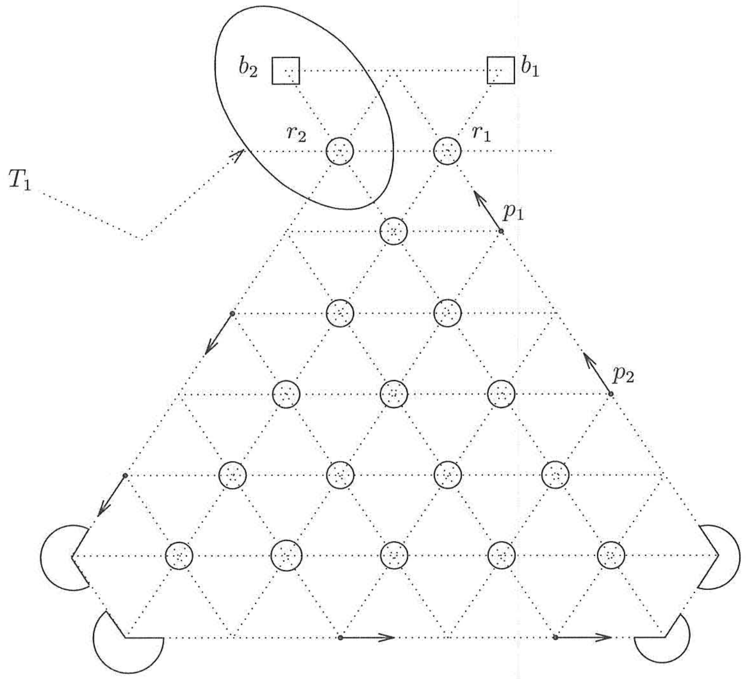

Figure 8: Pattern A detail. Particles p_1 and p_2 constitute a one-pair chain with velocity $\mathbf{v}_2$. Two type T_1 turns are shown in detail at the top of the figure: rest particle r_1 and barrier site b_1 constitute one turn, r_2 and b_2 constitute the other.

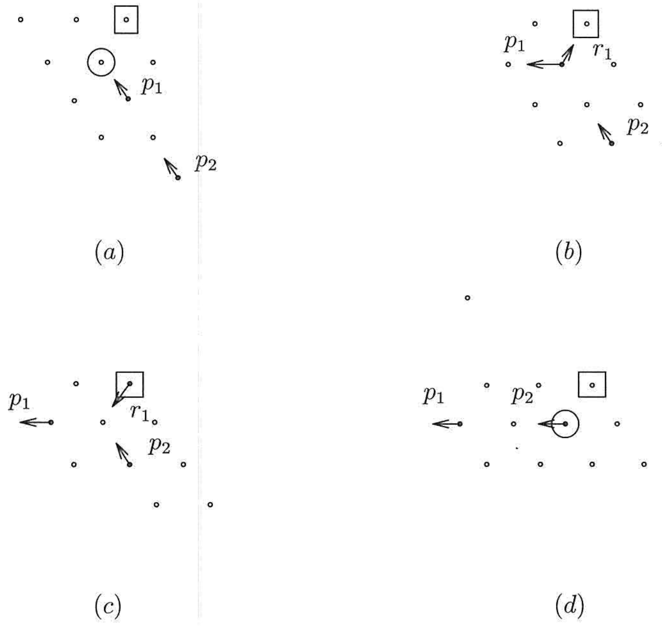

Figure 9: Four consecutive time steps of a pair of particles being deflected by a T_1 turn. (a) p_1 and the rest particle collide in a c_3 collision. (b) p_1 is deflected by $\pi/3$, r_1 collides with a barrier in a c_2^b collision. (c) p_2 and r_1 collide in a c_5 collision. (d) p_2 is deflected by $\pi/3$, r_1 is again at rest.

which we now describe in detail.

The T_1 turns used in A consist of a rest particle in the path of the oncoming stream, and a barrier at an adjacent lattice site. The location of the rest particle is called the *origin* of the turn: the incoming stream will experience a $\pi/3$ deflection from its path at the origin. Let us follow a pair of particles, p_1 and p_2, through a counterclockwise T_1 turn (see figure 9). The lead particle of the two-particle stream, p_1, collides with the rest particle, r_1, in a c_3 collision at $t = 0$. The resulting state of the origin has two particles leaving the site: one at $\pi/3$ counterclockwise from p_1's initial direction and one at $\pi/3$ clockwise, which we take to be p_1 and r_1, respectively. The barrier is located one site away from the origin, and at $t = 1$ particle r_1 experiences a c_2^b collision with the barrier sending r_1 back to the origin. At $t = 2$ particles r_1 and p_2 collide in a c_5 configuration at the origin, leaving r_1 in its original location at rest and sending p_2 out the same edge that p_1 used to exit the turn. The pair of particles have thus turned a corner, and any stream consisting of similar pairs can likewise be turned.

In this section we have shown the details of a single test pattern. The next task is that of constructing a test cycle from such a pattern, which

Confirmed One-Bit Error Coverage							
collision	error			collision	error		
	1–6	r	b		1–6	r	b
c_0	E_1	E_1	K_{27}	c_0^b	E_2		K_{27}
	E_1^*	K_{20}	E_1^*		E_2^*		E_2^*
c_1	E_3	K_{26}	E_3	c_1^b	E_4		K_{28}
	E_3^*	K_{21}	E_3^*		E_4^*		E_4^*
c_2	B_1	B_1	B_1	c_2^b	B_1		B_1
	B_1^*	K_{16}	B_1^*		B_1^*		B_1^*
c_3	A	A	A	c_3^b	G_2		G_2
	H_2^*	K_{17}	H_2^*		G_2^*		G_2^*
c_4	C_1	K_5	C_1	c_4^b	G_1		G_1
	C_1^*	K_6	C_1^*		G_1^*		G_1^*
c_5	A	A	A	c_5^b	G_6		G_6
	B_2^*	K_{18}	B_2^*		G_6^*		G_6^*
c_6	F_3	K_1	F_3	c_6^b	G_3		G_3
	F_3^*	K_2	F_3^*		G_3^*		G_3^*
c_7	C_2	K_7	C_2	c_7^b	G_1		G_1
	C_2^*	K_8	C_2^*		G_1^*		G_1^*
c_8	B_1	B_1	B_1	c_8^b	G_6		G_6
	B_1^*	K_{19}	B_1^*		G_6^*		G_6^*
c_9	F_1	K_3	F_1	c_9^b	G_3		G_3
	F_1^*	K_4	F_1^*		G_3^*		G_3^*
c_{10}	B_1	K_{12}	B_1	c_{10}^b	$D_1(cl)$		$D_1(cl)$
	B_1^*	K_{13}	B_1^*		$D_2^*(cl)$		$D_2^*(cl)$
c_{11}	B_1	K_{12}	B_1	c_{11}^b	$D_1(ccl)$		$D_1(ccl)$
	B_1^*	K_{13}	B_1^*		$D_2^*(ccl)$		$D_2^*(ccl)$
c_{12}	F_2	K_{10}	F_2	c_{12}^b	G_4		G_4
	F_2^*	K_{11}	F_2^*		G_7^*		G_7^*
c_{13}	$D_1(cl)$	K_{14}	K_{23}	c_{13}^b	G_5		K_{24}
	$D_1^*(cl)$	K_{15}	K_{22}		G_5^*		K_{25}

Table 1: For each collision each one-bit error was simulated in both dual and non-dual forms. The upper rows in each collision category are the non-dual cases, the lower rows are the duals.

Pattern A Collision Containment						
	period $\delta_A = 6$					
phase	0	1	2	3	4	5
collision	c_0 $c_{2(2,4,6)}$ c_1 $c_{3(1,3,5)}$	c_0 $c_{2(2,4,6)}$ $c^b_{2(1,3,5)}$ $c_{3(2,4,6)}$	c_0 $c_{2(2,4,6)}$ $c^b_{2(2,4,6)}$ $c_{5(1,3,5)}$	c_0 $c_{2(2,4,6)}$ c_1 $c_{5(2,4,6)}$	c_0 $c_{2(2,4,6)}$ c_1	c_0 $c_{2(2,4,6)}$ c_1

Table 2: The period of pattern A is adjustable by lengthening its chains; shown here is the containment for A with minimum period. Where more than one rotation of a collision occurs in a phase we have written the rotations in a list: for instance, $\{c_2(2), c_2(4), c_2(6)\}$ becomes $c_{2(2,4,6)}$.

is taken up in the following section. To do that we will need to know the error coverage of our patterns. By simulating the patterns as closed systems with a simulator that has its rule set altered by the corresponding error, we can test error coverage of the patterns. Table 1 shows the confirmed error coverage for all errors. From that table we can see that pattern A covers all one-bit errors for the non-dual c_3 and non-dual c_5 collisions. Another piece of information we will need is the collision timing table for our patterns. Table 2 shows the local time of each of the collisions c_3 and c_5 in pattern A.

6. Constructing a test cycle

This section shows the construction of a test cycle from several copies of pattern A. We have two goals in building test cycles: (1) to build a system that contains the complete collision class for one or more canonical collisions, and (2) to build a system that can present every processor in a multi-processor machine with collisions. The second goal can be achieved for our parallel machine, LGM-1, by including in the test ensemble patterns with shifted time origins, and we will return to this below. The first goal can be achieved by looking at the collisions contained in a pattern and adding enough symmetry transformed copies of the pattern to form a system that contains all the collisions of a particular class.

From the collision containment of pattern A listed in table 2, one can see that a single instance of A contains collisions c_0, c_1, c_2, c^b_2, c_3, and c_5. The dual of A, A^* would contain the duals of the collisions just mentioned. In pattern A, the canonical collision c_3 occurs in every rotation, $c_{3(1)}$ to $c_{3(6)}$, as do all the collisions in table 1 except c_2, which appears only in even rotations. Consequently, if we want a test cycle that covers all c_2 collisions we must combine A with a copy of A rotated by $\pi/3$. Let us call this combination test cycle Ω^1_A.

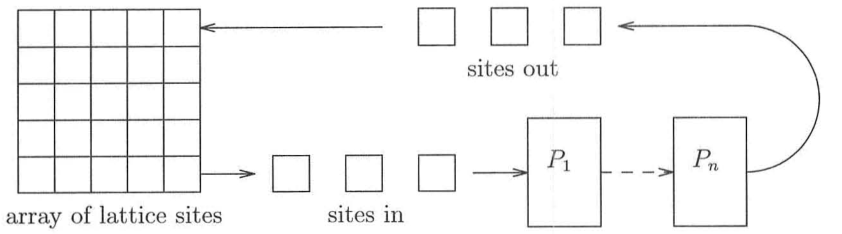

Figure 10: The LGM-1 architecture. The lattice sites are stored in an array and fed in raster scan order to a pipeline of processors. Each processor advances the lattice-gas system one evolution step and sends the data out in the same raster scan order.

The test cycle Ω_A^1 covers every rotation of the collisions listed in table 2. If we build a new test cycle Ω_A^2 by combining a copy of Ω_A^1 and a dual copy $(\Omega_A^1)^*$, we will have a test cycle containing the complete collision classes for every collision contained in pattern A. Unfortunately, Ω_A^2 does not achieve our goal for the collision class c_3. As we mentioned earlier, the c_3 and c_5 classes are each split into two because of dual asymmetry. As a consequence, pattern A does not cover any of the dual cases for c_3, but patterns H_2 and K_{17} together do, as can be seen in table 1. Consequently, a test cycle combining Ω_A^1 with test cycles built from these two additional patterns is required. Nevertheless, for the sake of simplicity in the following, we will assume that Ω_A^2 covers the c_3 and c_5 classes.

The second goal in test cycle construction involves collision containment for multi-processor implementations of the lattice-gas simulator. If the machine doing the lattice-gas simulation is a uniprocessor machine, we would be satisfied with this Ω_A^2 test cycle. Simulating this test cycle would cause the machine to repeatedly update the test cycle and every site would be processed by the same processor. Then, in a single period of the test cycle, the processor would "see" every collision in the collision classes contained in Ω_A^2. If, however, we are interested in testing a multi-processor machine, we need to insure that every processor sees every collision. We have been interested primarily in testing the LGM-1 machine, which is a linear pipeline in which each stage of the pipeline executes one update of the entire lattice-gas. In each stage there are two processors: one updates odd rows of the lattice, the other updates even rows. We next discuss the construction of a test cycle specifically targeted for the LGM-1 architecture (see figure 10).

For the moment we will assume each stage of LGM-1 has only one processor. If we look at the second stage of the pipeline, $P^{(2)}$, we see that it updates the automaton at global update steps $t_g \equiv 1(\text{mod N})$, where N is the number of stages in the pipeline. Consequently, in order for $P^{(2)}$ to see collision $c_{k(i)}$ we must build our test cycle so that at some $t_g \equiv 1(\text{mod N})$ we have $c_{k(i)}$ occurring somewhere in the cycle.

Because all the copies of pattern A in test cycle Ω_A^2 have the same phase, $t_{\text{local}}(0) = 0$, individual collisions are contained in some phases of Ω_A^2 and not others. For instance, from table 2 we see that the period of pattern A, δ_A, is six global update steps and that collision c_3 occurs in even rotations at local time $t \equiv 0(\text{mod } \delta_A)$, and odd rotations at $t \equiv 1(\text{mod } \delta_A)$. Therefore, a copy of Ω_A^2 with its local time origin set to coincide with the global time origin, $t_{\text{local}}(0) = 0$, has a period of six and contains all rotations of c_3 at $t \equiv 0(\text{mod } \delta_A)$ and at $t \equiv 1(\text{mod } \delta_A)$ Nevertheless, unless the period is relatively prime to N or $N/3$, every stage of the pipeline will not see c_3. One way to overcome this is to build a test cycle that contains every collision on every phase of its cycle. As table 2 shows, we can cover the missing cycle steps by reproducing Ω_A^2 three times with local time origins $t_{\text{local}}(0) = 0$, 2, and 4; the result we will call Ω_A^3. The complete test cycle Ω_A^3 containing six copies of pattern A has a cycle time of six and contains c_3 in every rotation at every phase of the cycle.

As we mentioned above, LGM-1 uses two processors within each stage; one processor handles the odd numbered rows and the other processor the even ones. To handle this we must check that our test cycle also contains every collision on both even- and odd-numbered rows. The simplest way around this problem is to make two copies of the test cycle, installing them so that their row positions differ by one. Alternately, one can ensure that every collision is represented for both row parities by adding more turns to pattern A to make a new pattern.

7. Error detection and experimental results

Our method of testing a simulator consists of simulating a complete test ensemble for many generations, and after the simulation, graphically displaying the ensemble system to detect errors in its evolution. For our machine, LGM-1, the state of a lattice-gas system is stored as a two-dimensional array of bytes in raster scan order, each byte containing the state of a single lattice site. Because we want to be confident that we are actually seeing the state of the system, we want the process of displaying it to minimize the amount of transformation done to the original data. We therefore treat the array as a bitmap graphics file and display it with a color map that color codes each byte.

The color coding is done so that, if there is only a single particle present at the site, the presence of the particle is indicated by a color corresponding to its velocity (see figure 11). If there are several particles present, their colors add in such a way that the resultant color corresponds to the vector sum of the velocities. Higher intensity corresponds to a greater number of particles present. So, for instance, if all six non-zero velocities are present, the site will appear bright white (white represents zero velocity); if no particles are present it will appear dim white; and if only a rest particle is present it will appear a noticeably brighter white than when no particle is present.

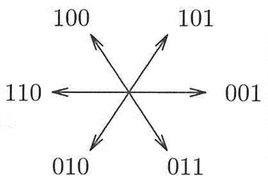

Figure 11: Color coding of unit vectors. Each direction vector is associated with an rgb (red, green, blue) color coding, and the zero vector is coded 111 (white). We imagine that the colors change continuously and linearly with the angle of rotation. For instance, $\mathbf{v}_1$ is associated with the triple 110, and $\mathbf{v}_2$ is associated with the triple 100. Any vector between $\mathbf{v}_1$ and $\mathbf{v}_2$ will have a triple of the form $\{1, x, 0\}$, where $0 \leq x \leq 1$. Given a collection of velocity vectors at a lattice site, adding the corresponding rgb triples and normalizing gives an rgb triple that codes a color that matches the resultant vector sum's direction. The number of particles present determines the brightness of the color. Thus $\mathbf{v}_1 + \mathbf{v}_2 \rightarrow 210$, and normalizing gives $\{1, 0.5, 0\}$ as the sum's color with a relative brightness of 2.

Different color maps can be used to bring out different features. For instance, in the color map described above, barrier sites are coded without color so that they appear black unless other particles are present at the site. If seeing the location of the barriers is important, the barriers can be coded with some distinctive color.

We want the presence of an error to result in a state of the system that is easy to distinguish visually from any of its correct states. Our patterns have the general property that an error disrupts the cycle and results in one or more particles straying beyond the sites occupied by the pattern. In most of our patterns we put a rest particle in any lattice site that is not part of the pattern but is inside the containing box of barriers. A stray particle passing into the space occupied by rest particles usually results in the rest particles erupting in a chain reaction that floods the box with moving particles. This chaotic state is visually unmistakable.

Figure 12 shows a small, example test cycle evolving in the presence of a one-bit error: the lattice-gas rules in a software simulator have been altered to contain the one-bit error $c_{3(1)} : e_1$. The test cycle consists of six copies of pattern A and is designed for periodic boundaries so that one of the patterns wraps around the sides. The pictures in the figure are the color-coded bitmaps of the states converted to gray scale: each pixel represents a lattice site, and the color scheme described above has been mapped to a gray scale.

In figure 12(a) the initial state of the test cycle is shown, and each succeeding image shows the system 10 time steps older. As is easily seen even in gray scale, the chaotic result is clearly distinguishable from the initial state. Because any legal state of the test cycle looks very similar to the

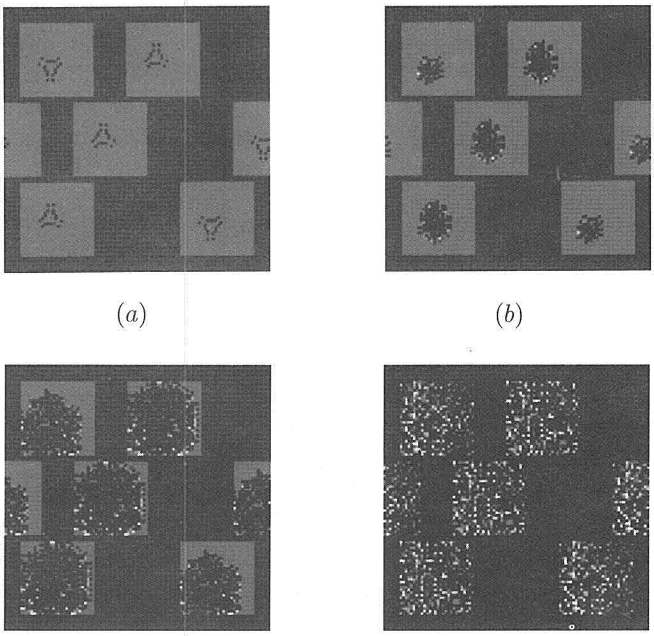

Figure 12: A test cycle containing six copies of pattern A simulated with the $c_3 : e_1$ error at every step. (a) The cycle in its initial state, $t = 0$. (b) $t = 10$. (c) $t = 20$. (d) $t = 100$. The second row of patterns in each picture above contains two patterns: one is "wrapped around" the left and right edges of the lattice.

initial state, we have no problem distinguishing the error indication from the system's correct states.

Besides having extension in evolution time, these test subsystems also have extension in the space of the lattice-gas, allowing them to detect errors such as those caused by incorrect data addressing by the lattice-gas simulator. For detecting these types of errors, the patterns that are "loops," such as pattern A, are useful. For instance, in LGM-1 a lattice-gas system is "cut" into strips that are fed to the pipeline one at a time and "sewn" together as they exit the pipeline. Placing a loop pattern across such strip boundaries makes it easy to see if there are any addressing errors made in the cutting and sewing operations. Likewise, if the boundaries of the lattice-gas are periodic, the loops can be placed across the boundaries. In LGM-1 the boundaries can be either periodic or not, and we test both cases with different test ensembles.

7.1 Detection difficulties

Our ensemble detects all significant one-bit errors, but not all are detected by chaotic conditions. One reason for this is the lack of dual symmetry for the collision of a single particle with a rest particle in the LGM-1 gas: we cannot have "explosions" in a dual world. We have two methods of getting around this.

One method is to build a pattern with many particles moving in an orderly fashion. Disruption of the pattern results in many stray particles and, while not giving the magnitude of chaos in an explosion, it is easy to see when the system fails to evolve correctly.

Another method is to enclose the dual pattern in a box with a device that functions as a gate for dual particles (holes). The dual particle is transformed to a real particle and the real particle can cause an explosion outside the dual box.

For some of our patterns, instead of employing either of these two methods we have relied on being able to see one or two stray particles or a global change that is not chaotic. While this is not the most desirable method of detecting errors, employing it allowed us to complete the set of patterns without spending time attempting to produce explosions for every error. Indeed, the attempt might be futile: it is an open question whether or not it is possible to find a collection of patterns that results in chaotic conditions for every one-bit error.

Another problem in detection is a consequence of the mismatch in scales caused by detection without chaos and the large number of patterns in our test ensemble. As we mentioned in section 4, our ensemble contains close to 1000 patterns. A chaotic change in any of 1000 30×30 pixel boxes displayed at one time is easy to see at a glance. But, because the size of the ensemble is much larger than the area of a single pixel (which may show the presence of a stray particle), the patterns that do not become chaotic require closer inspection. We have handled this by scanning a display of the test ensemble using a mouse-driven "magnifier" that allows us to see the patterns individually. While this works, it is not entirely satisfactory.

One way to approach this, besides trying to make all patterns detect by chaotic results, is to reduce the size of the ensemble. If an ensemble were small enough, a single particle would occupy a sufficient portion of the display so that, again, a glance would suffice. We have made no attempt to minimize our set. Rather, we have been interested in ensuring its completeness, and we have for the most part used each pattern as a test of a single canonical collision. As the description of test pattern A showed, a single pattern contains several collisions, and for this reason we estimate that our complete set is considerably larger than necessary.

For instance, the pattern that was specifically designed to test the c_0 (quiescent lattice site) collision is simply an empty box. If any particles or barriers appear in the box, the error is detected but the result is not necessarily chaotic. It seems likely that the c_0 test pattern is superfluous, and other patterns will detect any error that c_0 can. Confirming this could be done by simulating the eight possible one-bit errors for the canonical collision c_0 on the entire ensemble and checking to see that the errors were detected by some other pattern in every case.

There is one type of one-bit error that we make no claim of detecting. Because these errors do not affect the lattice-gas behavior in a way that changes its modeling capability, we consider these types of errors insignificant. As table 1 shows, these errors are all rest-particle errors in collisions in which a rest particle and a barrier exist at the same site. As we mentioned when we described the LGM-1 rule set, in the LGM-1 gas the rest particle disappears in these configurations. Consequently, any test pattern for this type of collision can only be used as a one-shot test, and this test must occur at $t = 0$.

The patterns we have devised for these collisions do detect all one-bit errors, but the one-shot nature of this testing forces us to deny fully covering these collisions for two reasons: one is that the errors cannot be detected in any processor other than the first one in an LGM-1 type pipeline machine, and the other is that the patterns must be inspected very closely to detect the error of a rest particle remaining at the site after $t = 0$. In general, this is a difficult type of change to detect because the rest particle at a barrier lattice site has no interaction with other particles. Altogether, 28 of our test patterns, K_1 through K_{28}, were created especially to deal with these rest-particle problems.

7.2 Experimental results

In this section we describe the results of experiments with simulated multi-bit errors. The experiments we performed for the c_3 collision show it is unlikely the ensemble will fail to detect multi-bit errors. Using a software simulator for the LGM-1 gas, we ran simulations of the test cycle in figure 12. For each simulation run we altered the rule set to include an error in the result of the $c_{3(1)}$ collision. A total of 117 simulations were run, each for 20 update steps: all 8 possible one-bit errors, all 28 possible two-bit errors, all 56 possible three-bit errors, and 25 selected four-bit errors. The four-bit errors were all those of the form $e_{3,5,x,y}$ and $e_{5,r,x,y}$ because these were the types of errors that resulted in detection difficulties in the two-bit and three-bit cases. After each simulation the state of the test cycle was observed using the technique described in section 7.1. As table 3 shows, there was only one error that failed the test completely, namely the $c_{3(1)} : e_{5,r}$ error.

The reason the $e_{5,r}$ error escaped detection is that in this test cycle the error partially erases its own mistake in such a way that the cycle continues undisturbed. Of course, a careful inspection of the individual pixels would reveal the error, but from our viewpoint this is not an acceptable method of error detection. Rather, we want the error to expose itself in such a way that a glance at a monitor would suffice to determine its existence. One remedy for this particular detection problem is to make a copy of pattern A with the

Experimental Results for Simulated Errors					
number of incorrect bits		1	2	3	4
non-chaotic detection results	stray particles	0	$e_{3,5}$	$e_{3,4,5}$	0
	lack of particles	0	0	$e_{3,5,r}$	$e_{3,4,5,r}$
	undetected	0	$e_{5,r}$	0	0

Table 3: The results of experiments simulating multi-bit errors in the evolution of the test cycle shown in figure 12. All errors simulated resulted in a chaotic system except those listed above. For instance, the two-bit error $e_{3,5}$ resulted in "stray particles" occurring in the test cycle. Stray particles are defined as particles in the test cycle that are moving beyond the limits of the particle paths in the correctly functioning cycle. A "lack of particles" result means that all the moving particles have dissappeared from the cycle. An "undetected" result means that the cycle is visually indistinguishable from a correctly evolving test cycle. This last result could be detected by a program that directly compares two cycles.

particle chains traveling around the loop in the opposite direction. The new pattern would detect the $c_{3(1)} : e_{5,r}$ error but would fail for the $c_{3(4)} : e_{6,r}$ error, which one would expect since the two patterns are mirror images of each other, as are the two errors just mentioned.

8. Architectures and applicability of the test method

The testing method we have described was conceived with the LGM-1 pipeline architecture in mind. Coincidently, the method also works well for testing software implementations. The element that makes the method applicable to LGM-1 is the scanning style of data flow through the update processors. Because of this data scanning, the entire lattice is passed through every processor, every collision contained in the lattice-gas system is processed by every processor, and thus all the update logic is tested.

Contrasted to this scanning data flow is the parallel implementation of a lattice-gas automaton that assigns a single processor to each lattice site. Here, the test method described in this paper implies building a lattice-gas system that has every collision occurring at every lattice site. This is probably impractical for two reasons, one of which is that the test ensemble would be excessively large. For instance, one would need a separate lattice-gas system for every possible collision, which amounts to $28 \times 6 \times 2 = 336$ lattice-gas systems the size of the entire automaton. Which brings us to the second reason this is a bad idea: even if the set of test patterns was much smaller, the detection would have to be done without the aid of chaotic "explosions" contrasting with an ordered system. While it is true that an entire lattice initialized to contain the same collision at every lattice site may appear organized to start with, detecting the failure of a small percentage of randomly located processors would probably require looking closely at every lattice site because the lattice-gas system would likely appear randomized after a few updates.

Between the the architecture mentioned above using one processor per lattice site and the architectures that scan the entire lattice—such as uniprocessors and linear pipelines like LGM-1—lies a continuum of architectures we dub "frame-oriented architectures." A frame-oriented architecture processes a lattice by assigning the update of some fixed region, or frame, of the lattice to each processor and exchanges information about the frame boundaries between appropriate processors. Our test method is directly applicable when the number of frames is small because we can duplicate the test ensemble for each frame and proceed as usual, treating each frame as if it were handled by a separate machine. The LGM-1 machine is a two-frame architecture because the lattice sites in even and odd rows are processed by what amounts to two separate pipelines of processors. As we said earlier, we handle this by duplicating the test ensemble and translating it by one row in the lattice. In general, this approach is only worthwhile when the number of frames is small, and consequently this presents a trade-off in the design of architectures for lattice-gas simulations. That is, one must trade testability, at least by our method, against the number of frames the architecture employs.

9. Summary and conclusions

We have shown how to test lattice-gas simulators with reasonable resources. Easily discernible changes in cyclic "particle" patterns signal an evolution error in the lattice gas, and a brief visual scan of bitmapped graphics suffices to determine whether or not an error has occurred. More precisely, our test ensembles are built from a library of 51 pattern templates, each pattern occupying roughly 50 lattice sites contained in a "box" of barrier sites with 30 lattice spacings on a side. A naïvely constructed ensemble to test a software simulator of an LGM-1 gas requires 612 patterns: two duals and six rotations of all 51 pattern templates. The ensemble we have used for this purpose consists of 393 patterns. An ensemble for the two-frame architecture of the LGM-1 requires twice that many, or 786 patterns. The display of this 786 pattern ensemble occupies about 77% of the host's display. For the purposes of testing every processor in an LGM-1 type architecture containing an arbitrary number of stages, time-shifted copies of the patterns are required, resulting in an ensemble about three times larger, that is, containing about 2400 patterns displayed on two and one-half screens of bitmapped graphics. In practice, we have always set the number of pipeline stages so that the number of stages is relatively prime to the cycle times of the patterns, and consequently only 786 patterns are needed for our complete ensemble. Because of the hole/particle symmetry of the FHP-III gas, a complete ensemble for it requires only about two-thirds the number of patterns as the LGM-1 gas.

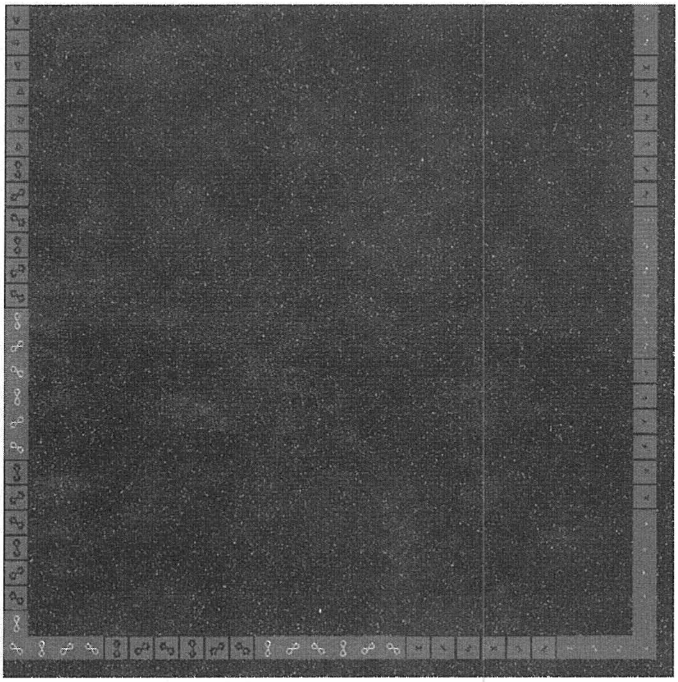

Figure 13: Test patterns embedded in an 800 × 800 site lattice-gas fluid flow experiment. The state of the lattice is represented by a gray scaled version of the color scheme mentioned in section 7. The light gray border on three sides contains 76 test patterns. The stippled center section contains the lattice-gas flow, seen here after 10,000 update generations. Although it cannot be seen, there is a forced flow across the top edge of the image. The test patterns are incorporated into the boundaries defining the shape of the flow "well."

These ensembles detect all significant one-bit errors in the evolution of a simulated lattice-gas system, and our experiments suggest that multiple errors are unlikely to escape detection. Simulating the complete ensemble tests the correctness of the implementation of the update rules, the data addressing logic, and data transmission and general system functions to the extent that they affect the lattice-gas system's evolution. Because the patterns are cyclic, testing continues for as long as the lattice-gas system containing them evolves. This lets the ensembles test every processor in a pipelined architecture such as LGM-1, and it also lets them act as runtime error detectors by incorporating them into real fluid flow lattice-gas systems. Also, these embedded tests allow some detection of transient failures in the simulating system during simulation runs.

The technique described in this paper can be used in any software implementation. For hardware implementations the technique's applicability depends on the way the lattice sites are assigned to the processors. If the lattice is split into non-overlapping pieces, or frames, that are updated by disjoint sets of processors, the ensemble must be duplicated for each such frame. As the number of frames increases the technique soon becomes impractical, and there is consequently a trade-off between testability by this method and the architectural parameter associated with the number of frames.

Of course, as with any testing facility, the issue of the correctness of the thing being tested becomes the issue of the correctness of the test. For our test method, the issue becomes one of verifying the correct implementation of the test patterns, and the question then becomes one of establishing the correctness of the software, amounting to several thousand lines of code, that constructs and manipulates the test ensemble. While confirming that the software is free of bugs by traditional software testing and verification methods is difficult, we have been able to refer to the resulting patterns themselves for confirmation. The reason is that the patterns are simple and easy to understand visually, even when displaying them amounts to no more than color coding an octal dump of the data. Thus, we have been able to use the patterns themselves to debug the software that creates them.

The use of this test method in practice has shown it to be an efficient aid in the construction, verification, and operation of lattice-gas simulators. In the construction of the machines we have used it to screen for faulty custom chips that contain the processors of the LGM-1 lattice-gas simulator, and to detect hardware design errors in the custom boards. For these purposes we have used a subset of the complete set of patterns and found that screening a single chip requires about three minutes of real time. Most of this time is spent inserting the chip in the test socket, setting up the simulation, and displaying the result. We have used the test method during construction of software controlling the simulation. We found that writing simulation software was speeded up considerably by the availability of a concurrent test facility: the code could be quickly written, modified, or redesigned because a simple five-minute test would detect errors as they were introduced into the system and let them be corrected immediately. In fact, the test template library routines, the software simulator, and the image handling library were developed in parallel: the simple graphical nature of the test patterns allowed debugging of each concurrently, even though none of these systems was complete. For verification of the system before and after simulation runs, the entire system test required about five minutes of real time. We have also conducted fluid flow experiments on LGM-1 using an 800×800 lattice containing embedded test patterns to detect runtime errors (see figure 13). These embedded test patterns added about 10% to the simulation time, and the complexity of specifying the initial state of the lattice was increased by approximately a factor of two.

Acknowledgement

We would like to thank David P. Dobkin and Eleftherios Koutsofios for their encouragement, advice, and assistance in the graphic display of the test patterns and ensembles. The displays were performed on SUN 3/160C and IRIS 4D 220GTX workstations[2] using the *Cheyenne* [23] graphics library. This work was supported in part by NSF Grant MIP-8912100, and U.S. Army Research Office—Durham Contract DAAL03-89-k-0074.

References

[1] U. Frisch, B. Hasslacher, and Y. Pomeau, "Lattice-gas Automata for the Navier-Stokes Equation," *Physical Review Letters*, **56** (1986) 505–1508.

[2] D. d'Humiéres, P. Lallemand, and U. Frisch, "Lattice Gas Models for 3D Hydrodynamics," *Europhysics Letters*, **4**(2) (1986) 291–297.

[3] U. Frisch, D. d'Humiéres, B. Hasslacher, P. Lallemand, Y. Pomeau, and J. P. Rivet, "Lattice Gas Hydrodynamics in Two and Three Dimensions," *Complex Systems*, **1** (1987) 649–707.

[4] D. d'Humiéres, P. Lallemand, and T. Shimomura, "Lattice Gas Cellular Automata: A New Experimental Tool for Hydrodynamics," Preprint LA-UR-85-4051, Los Alamos National Laboratory, Los Alamos, New Mexico (1985).

[5] J. Salem and S. Wolfram, "Thermodynamics and Hydrodynamics with Cellular Automata," *Theory and Applications of Cellular Automata*, vol. 1, edited by S. Wolfram (World Scientific, Singapore, 1987).

[6] F. Hayot, M. Mandal, and P. Sadayappan, "Implementation and Performance of a Binary Lattice Gas Algorithm on Parallel Processor Systems," *Journal of Computational Physics* (1987).

[7] T. Shimomura, G. D. Doolen, B. Hasslacher, and C. Fu, "Calculations Using Lattice Gas Techniques," *Los Alamos Science*, **15** (1987) 201–210.

[8] D. d'Humiéres and P. Lallemand, "Numerical Simulations of Hydrodynamics with Lattice Gas Automata in Two Dimensions," *Complex Systems*, **1** (1987) 599–632.

[9] L. Kadanoff, G. McNamara, and G. Zanetti, "A Poiseuille Viscometer for Lattice Gas Automata," *Complex Systems*, **1** (1987) 791–803.

[10] H. Lim, "Lattice Gas Automata of Fluid Dynamics for Unsteady Flow," *Complex Systems*, **2** (1988) 45–58.

[11] H. Chen, S. Chen, G. Doolen, and Y. C. Lee, "Simple Lattice Gas Models for Waves," *Complex Systems*, **2** (1988) 259–267.

[2]IRIS is a trademark of Silicon Graphics, Incorporated.

[12] R. Benzi and S. Succi, "Bifurcations of a Lattice Gas Flow under External Forcing," *Journal of Statistical Physics*, **56**(1/2) (1989).

[13] P. Binder, "Abnormal Diffusion in Wind-tree Lattice Gases," *Complex Systems*, **3** (1989) 1–7.

[14] A. Clouqueur and D. d'Humiéres, "RAP1, a Cellular Automata Machine for Fluid Dynamics," *Complex Systems*, **1** (1987) 585–597.

[15] N. Margolus and T. Toffoli, "Cellular Automata Machines," *Complex Systems*, **1** (1987) 967–993.

[16] S. D. Kugelmass and K. Steiglitz, "A Scalable Architecture for Lattice-Gas Simulations," *Journal of Computational Physics*, **84** (1989) 311–325.

[17] T. J. Ostrand and M. J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," *Communications of the ACM*, **31**(6) (1988) 676–686.

[18] D. Gelperin and B. Hetzel, "The Growth of Software Testing," *Communications of the ACM*, **31**(6) (1988) 687–695.

[19] P. W. Kasteleyn, "Graph Theory and Crystal Physics," in *Graph Theory and Theoretical Physics*, edited by F. Harary (Academic Press, New York, 1967).

[20] K. Preston, Jr., and M. J. B. Duff, *Modern Cellular Automata: Theory and Applications* (Plenum Press, New York, 1984).

[21] T. Toffoli and N. Margolus, *Cellular Automata Machines: A New Environment for Modeling* (MIT Press, Cambridge, 1987).

[22] R. Squier and K. Steiglitz, "Testing Parallel Simulators for Two-Dimensional Lattice-Gas Automata," Technical Report CS-TR-269-90, Princeton University, Computer Science Department, 35 Olden Street, Princeton, NJ 08544-2087, (June 1990).

[23] D. P. Dobkin and E. Koutsofios, "The Cheyenne Graphics Library," unpublished internal documentation, Computer Science Department, Princeton University (1989).