

# A READER'S GUIDE TO GACS'S "POSITIVE RATES" PAPER

LAWRENCE F. GRAY

ABSTRACT. Peter Gacs's monograph, which follows this article, provides a counterexample to the important Positive Rates Conjecture. This conjecture, which arose in the late 1960's, was based on very plausible arguments, some of which come from statistical mechanics. During the long gestation period of the Gacs example, there has been a great deal of skepticism about the validity of his work. The construction and verification of Gacs's counterexample are unavoidably complex, and as a consequence, his paper is quite lengthy. But because of the novelty of the techniques and the significance of the result, his work deserves to become widely known. This reader's guide is intended both as a cheap substitute for reading the whole thing, as well as a warm-up for those who want to plumb its depths.

Peter Gacs's paper tells us something mathematically rigorous about phase transitions that was thought by many to be impossible. Namely, it contains an example of a one-dimensional lattice system that has a phase transition, in spite of the fact that it only has nearest-neighbor translation-invariant interactions and a finite local state space. To be sure, his model differs from standard statistical mechanical models, such as the Ising model, in that it is not defined in terms of a Hamiltonian. But it is not clear why that makes a difference.

In any case, the highly organized cooperative behavior found in Gacs's model is worthy of study. Unfortunately, the paper is complicated enough so that very few will actually want to work their way through the details. It is my intent to provide you with a simplified version, with as little of the messiness as possible. Naturally, my presentation will be lacking somewhat in rigor. Nevertheless, I hope it will give you an understanding of all of the key ideas.

## 1. THE MATHEMATICAL SETTING

In this section I will try to lay the mathematical groundwork for our discussion as efficiently and painlessly as possible. In particular, I need to explain the idea of a cellular automaton with random errors. In Section 2, I will give a few simple examples and explain the connection to statistical mechanics. The notation and terminology introduced here will be consistent with that used by Gacs, except for some minor modifications.

Underlying all of the models to be discussed here is a discrete *lattice*  $\Lambda$ , and in all of our examples, we will take  $\Lambda$  to be either the set of integers  $\mathbb{Z}$ , or the set of integer pairs  $\mathbb{Z}^2$ . These two options are called, respectively, the "one-dimensional case" and the "two-dimensional case". Each member of the lattice  $\Lambda$  is called a *site*.

---

*Key words and phrases.* phase transition, universal computation, error-correction, cellular automaton, stability, self-simulation.

Also associated with each model is a *state space*, typically denoted by  $\mathbb{S}$ . In all cases to be considered here,  $\mathbb{S}$  will be a finite set. This set represents the possible states at a site. In applications to statistical mechanics, a common choice is  $\mathbb{S} = \{-1, +1\}$ . For example, in the famous “Ising model”, the lattice represents an array of electrons, and the states correspond to two possible spin directions, “spin down” and “spin up”.

If we specify a state for each site of the lattice, we obtain a *configuration*. The set of all configurations is thus the set  $\mathbf{C} = \mathbb{S}^\Lambda$ . Generically, an element of configuration space will be denoted by the Greek letters  $\xi$  or  $\eta$ . The state taken by a configuration  $\xi$  at a site  $x$  is denoted by  $\xi(x)$ . Configurations can also depend on time, in which case they are called *space-time configurations* and typically denoted by an expression like

$$\xi(x, t),$$

which stands for the state taken by the space-time configuration  $\xi$  at site  $x$  and time  $t$ . To keep things simple in this article, I will always take time to be discrete, in spite of the fact that for most applications to statistical mechanics, continuous time is the more convenient choice. For the kinds of qualitative behavior that I want to discuss, the difference between discrete and continuous time is not important. Gacs’s model can be implemented in both settings.

Given a space-time configuration  $\xi$ , a set  $A \subseteq \Lambda$ , and a time  $t$ , we will sometimes abuse notation slightly, and write

$$\xi(A, t)$$

for the restriction of  $\xi(\cdot, t)$  to the sites in  $A$ .

Now I will introduce some deterministic dynamics. Only two ingredients are needed: an *interaction neighborhood*  $\mathcal{N}$ , which is simply a finite subset of  $\Lambda$ , and a *transition function*  $\text{Tr}$ , which is a function from  $\mathbb{S}^{\mathcal{N}}$  to  $\mathbb{S}$ . These ingredients determine a *cellular automaton*, which is the collection of space-time configurations  $\eta$  that satisfy the following condition:

$$(1.1) \quad \eta(x, t) = \text{Tr}(\eta(\mathcal{N} + x, t - 1)), \quad x \in \Lambda, t > 0,$$

where  $\mathcal{N} + x = \{y + x : y \in \mathcal{N}\}$ . Each such  $\eta$  is called a *trajectory* of the cellular automaton. Clearly, there is precisely one trajectory  $\eta$  for each *initial configuration*  $\eta(\cdot, 0)$ .

Next I want to introduce random “errors” into the model. In addition to the two ingredients that define a cellular automaton, I need two further ingredients: a parameter  $\varepsilon \in [0, 1]$ , called the *error rate*, and a probability measure  $\nu$  on the state space  $\mathbb{S}$ . I call this probability measure the *noise distribution*. I will be particularly interested in the case where  $\nu$  puts positive mass on every member of  $\mathbb{S}$ . When this condition holds, I will say that  $\nu$  is *strictly positive*.

The error rate is used to define a *random error set*  $E$ . This is a random subset of  $\{(x, t) : x \in \Lambda, t > 0\}$  whose probabilistic properties are determined by the condition that the events

$$\{(x, t) \in E\}, \quad x \in \Lambda, t > 0,$$

be independent, with each individual event having probability  $\varepsilon$ . The noise distribution is used to define a collection of  $\mathbb{S}$ -valued random variables  $s(x, t)$ ,  $x \in \Lambda, t > 0$ , called *noise variables*. The probabilistic properties of the noise variables are determined by the condition that they be independent of each other and of  $E$ , and that they all have distribution  $\nu$ .

Given an interaction neighborhood  $\mathcal{N}$ , a transition function  $\text{Tr}$ , a random error set  $E$ , and a collection of noise variables  $s(x, t)$ , the corresponding *faulty cellular automaton* is a family of stochastic processes, one for each initial configuration. Each stochastic process in this family is a random space-time configuration  $\eta$  determined by three things: (i) the initial configuration  $\eta(\cdot, 0)$ ; (ii) the requirement that (1.1) hold for all  $(x, t) \notin E$ ; (iii) the requirement that  $\eta(x, t) = s(x, t)$  for  $(x, t) \in E$ . Each such  $\eta$  is called a *random trajectory* for the faulty cellular automaton. The initial configuration of a random trajectory can itself be random, in which case I assume that it is independent of  $E$  and of the collection of noise variables. It is not hard to check that all of the random trajectories in a faulty cellular automaton are discrete-time Markov processes, each having the same transition probabilities.

The informal description of a faulty cellular automaton is this. The system follows the transition rule  $\text{Tr}$  everywhere except at space-time points  $(x, t) \in E$ . For  $(x, t) \in E$ , independently choose a value for  $\eta(x, t)$  using the noise distribution. I will say that an *error* occurs at  $x$  at time  $t$  for each  $(x, t) \in E$ , even if the value of  $\eta(x, t)$  happens to agree with the value that would have resulted from applying (1.1) at  $(x, t)$ . Note that the occurrence of an error can affect states at points  $(x, t) \notin E$ , by way of (1.1).

Suppose  $\eta$  is a random trajectory for a faulty cellular automaton. If the probability distribution of  $\eta(\cdot, t)$  is independent of  $t$ , then that probability distribution is called a *stationary measure* for the faulty cellular automaton. I will now state the Positive Rates Conjecture, which for simplicity I give in the context of faulty cellular automata. There are versions of the conjecture that refer to more general systems, such as “probabilistic cellular automata”, and their continuous-time analogues, “interacting particle systems”. Gacs’s counterexample applies to all such versions.

**Conjecture** (Positive Rates). *A one-dimensional faulty cellular automaton with (i) finite interaction neighborhood  $\mathcal{N}$ , (ii) strictly positive error rate  $\varepsilon$ , and (iii) strictly positive noise distribution  $\nu$ , can have at most one stationary measure.*

The phrase “positive rates” refers to conditions (ii) and (iii). It is not uncommon to be somewhat casual with terminology from statistical mechanics and say that a system has a phase transition if it has positive rates and more than one stationary measure. So a brief version of the conjecture is “no phase transitions in one dimension”. This statement is known to be true for the restricted class of legitimate statistical mechanics models with finite-range interactions, where “legitimate” means that the dynamics are defined in terms of a Hamiltonian function according to a standard procedure (see [7] for details).

## 2. EXAMPLES AND THE PHYSICS CONNECTION

The following examples will help to explain the connection between cellular automata and statistical mechanics. They will also help prepare you for the more elaborate model that constitutes the main point of this article.

A key feature in each of these examples is the presence of *ground states*. A ground state of a cellular automaton is a configuration  $\xi$  that is a fixed point of the dynamics. That is,  $\text{Tr}(\xi(\mathcal{N} + x)) = \xi(x)$  for all  $x \in \Lambda$ . The most common types of phase transitions have to do with ground states.

*Example 1* (Majority vote). For this model, the state space is the same as for the Ising model, namely  $\mathbb{S} = \{-1, 1\}$ . I will consider both  $\Lambda = \mathbb{Z}$  and  $\Lambda = \mathbb{Z}^2$ . For both

choices of  $\Lambda$ , I will assume *nearest-neighbor interactions*. That is,

$$\mathcal{N} = \begin{cases} \{-1, 0, 1\} & \text{if } d = 1 \\ \{(-1, 0), (0, -1), (0, 0), (1, 0), (0, 1)\} & \text{if } d = 2. \end{cases}$$

Transitions are determined by taking a simple majority vote:

$$\text{Tr}(\xi(\mathcal{N} + x)) = \text{the majority value in } \{\xi(y) : y \in \mathcal{N} + x\}.$$

For the random model, I also need to specify the noise distribution. To preserve  $(+/-)$ -symmetry, I let  $\nu[-1] = \nu[+1] = 1/2$ .

This model clearly has many ground states. The two most important ones are “all +1” and “all -1”. Imagine starting the model with the initial configuration “all +1”. Introduce a random error set with a small noise parameter  $\varepsilon > 0$ . You might expect that, in spite of the errors, the majority vote mechanism will prevent the system from straying very far from its initial configuration. This expectation is justified for the two-dimensional case but not for the one-dimensional case, as I will explain.

First, let me talk about the Ising model, for the sake of comparison. It has the same configuration space as the majority vote model. It also has the two ground states “all +1” and “all -1”. In the dynamical version of the Ising model (see [7]), the Hamiltonian leads to a kind of weighted majority vote transition function that is qualitatively like the transition function of the model I have been describing. The Ising model also has a kind of noise parameter, called “temperature”. The “errors” in the Ising model are sometimes called “thermal fluctuations”.

The behavior of the Ising model near the ground states depends on the dimension of the lattice. In two dimensions, it can be mathematically proved that when the temperature is positive and sufficiently small, there is a stationary measure “near” each ground state. So there is a phase transition. In one dimension, the model tends toward a unique “fifty-fifty” stationary measure, no matter what the initial configuration is; there is no phase transition in one dimension.

The majority vote cellular automaton is believed to behave qualitatively like the dynamic Ising model as far as the ground states “all +1” and “all -1” are concerned. Unfortunately, only the one-dimensional result has been shown mathematically (see [5]). For the two-dimensional result, we only have computer simulations. But no one seriously believes that there are any surprises here. I am quite confident myself that the majority vote model has a phase transition in two dimensions.

Symmetry is important for this model. If  $\nu$  is changed, thereby favoring one of the ground states, then it is believed that the system has a unique stationary measure for any  $\varepsilon > 0$ . This is analogous to what happens when there is an external field in the Ising model. The phase transition seems to require “symmetric errors”.

□

*Example 2* (Discrete-time contact). The state space is  $\mathbb{S} = \{0, 1\}$ . I consider only the one-dimensional, one-sided, nearest-neighbor version:  $\Lambda = \mathbb{Z}$  and  $\mathcal{N} = \{-1, 0\}$ . The transition function is given by

$$\text{Tr}(\xi(x-1), \xi(x)) = \max\{\xi(x-1), \xi(x)\},$$

where I have chosen to write  $\text{Tr}(\xi(x-1), \xi(x))$  rather than  $\text{Tr}(\xi(\mathcal{N} + x))$  because the interaction neighborhood is so small. The phrase “one-sided contact” refers to the fact that the 1’s in this model spread towards the right “by contact” with the

nearest neighbor. For the noise distribution  $\nu$ , I let  $\nu[0] = 1$  and  $\nu[1] = 0$ . A full discussion of the continuous-time version of this model can be found in [7].

This model has two ground states, “all 0” and “all 1”. Because of the biased noise, “all 0” is clearly a fixed point for the dynamics, even when  $\varepsilon > 0$ . So the point mass at “all 0” is a trivial stationary measure for the system. It can be shown that for sufficiently small  $\varepsilon > 0$ , the system also has a (nontrivial) stationary measure close to “all 1”.

Is this a phase transition? Not really, because my choice for  $\nu$  was not strictly positive. It can be proved mathematically that if we make any other choice for  $\nu$ , then for small  $\varepsilon > 0$ , the system will go to a unique stationary measure that is near “all 1” no matter what the initial configuration is.

One could say that the “all 1” ground state is *stable* under strictly positive noise perturbations for this model, while the “all 0” ground state is not. The stability of the “all 1” ground state does not depend on the choice of  $\nu$ . The ground states in the majority vote and Ising models also have a kind of stability under a strictly positive noise perturbation, but only when  $\nu$  is symmetric.  $\square$

*Example 3* (Toom). This model is the same as the two-dimensional majority vote model, except for one difference: the interaction neighborhood is reduced in size to  $\mathcal{N} = \{(1, 0), (0, 1), (0, 0)\}$ . So the majority vote is taken over the so-called “NEC neighborhood”, consisting only of the “Northern”, “Eastern”, and “Center” sites of the set of nearest neighbors.

The Toom model has the ground states “all +1” and “all -1”. Toom [9] proved that both of these ground states are stable under noise perturbations, for all choices of the noise distribution  $\nu$ . So the Toom model has a phase transition, and in a sense, it is a stronger one than the phase transition in the majority vote and Ising models, because symmetry of the noise distribution is not required. The phase transition in Gacs’s model is of this type.

There is no natural one-dimensional version of the Toom model. If you try to make one, you will get a variation on either the one-dimensional discrete-time contact model, or the one-dimensional majority vote model, and the behavior will be qualitatively like that of those models.  $\square$

All of the examples given above share two important common features. The first feature is *monotonicity*, which roughly says that if you change the configuration by switching some of the -1’s to +1’s, then in a certain natural sense, you improve the chances for the appearance of +1’s in the future evolution of the system. In statistical mechanics, such monotone models are said to be *ferromagnetic*. The standard Ising model is ferromagnetic, but there are generalizations, such as the so-called “spin-glass” models, that are not. The other common feature is that the various stationary measures are associated with ground states. Most of the phase transitions in statistical mechanics are related to ground states in some way.

A thorough study of these examples suggests the following intuitive picture. Imagine that you start with the ground state “all +1” in one of these models, and then let the noise do its work. This noise will repeatedly produce arbitrarily large finite islands of -1’s within the sea of +1’s. If the “all +1” ground state is to be stable, these islands of errors must be kept under control. If the “all -1” ground state is stable as well, then the dynamics must also keep control over finite islands of +1’s in a sea of -1’s. The Toom and majority vote models show how this can happen in two-dimensional models. Roughly speaking, finite islands in

two dimensions contain extra “corners” that can be “eroded” by the surrounding sea. But in a one-dimensional model, it would seem impossible for any dynamics with finite interaction neighborhood to distinguish between a large island and the sea surrounding it. At the interface between an island of one type and a sea of the opposite type, the dynamics would only “see” an interface between two seas of opposite type. How can both ground states be stable?

Although this argument may seem too simple to deal with something as general as the Positive Rates Conjecture, it has in fact formed the basis for results that support that conjecture. For example, the remarkable stability results of Toom [9] are based on the idea of eliminating finite islands, and his results imply that a one-dimensional monotone model cannot have two ground states that have the same strong kind of stability that the two-dimensional Toom model has. And I proved a result in [4] that supports the conjecture, using a form of the finite islands idea. I think it is fair to say that anyone who has had reason to believe the conjecture has some form of this argument in mind. I also think that it is not too much of an oversimplification to say that understanding the counterexample of Gacs basically amounts to understanding just how it takes care of arbitrarily large finite islands of errors.

### 3. A BRIEF HISTORY OF THE PROBLEM

I first became aware of the Positive Rates Conjecture when I was a graduate student at Cornell, working under Frank Spitzer in the mid-1970’s. At that time we heard rumors that a Russian logician had found a strange counterexample. The logician was Kurdyumov, and his example was said to be based on ideas from logic, especially the concept of a universal Turing machine. Eventually, my colleague and fellow graduate student David Griffeath obtained a copy of a short description of this counterexample, written by Kurdyumov ([6]). Unfortunately, this article seemed to contain no mathematics whatsoever. Instead, it was a description of some scenario involving colonies of workers, with defectors infiltrating neighboring colonies, and other such things. David called it a “spy novel”. We also heard that when Kurdyumov tried to explain his counterexample to experts in the field (such as Dobrushin), there was no one who understood enough of what he said to be able to pass judgment on its validity. We eventually decided that the counterexample was at best a mathematical fantasy.

Shortly afterward, Kurdyumov teamed up with Gacs and Levin [3] to produce a very simple one-dimensional model with state space  $\{-1, +1\}$  and range-2 interactions. The model has ground states “all +1” and “all -1”, and they conjectured that these ground states are stable under noise perturbation. No proof of stability was given, but computer simulations are quite convincing. Furthermore, when the error rate is 0, the (deterministic) dynamics of the model are such that finite islands of one type are always eliminated by a sea of the opposite type. However, this elimination of finite islands happens in a fragile manner, and from the very beginning, many of us doubted whether there was any type of stability. Recent results of Park [8], a student of Gacs, give strong evidence that the ground states are in fact unstable in this model.

On the opposite side of the problem, I proved in [4] that for monotone, one-dimensional, nearest-neighbor, continuous-time models having the state space  $\mathbb{S} = \{-1, +1\}$ , only one stationary measure exists if the rates are positive. Note that

this result concerns general stationary measures, not just those that might result from perturbing ground states. It is still an open question whether this result holds true when the nearest-neighbor assumption is relaxed to finite range interactions. Gacs's model is not monotone, so it does not answer this question.

Fortunately, Gacs did not give up on finding a counterexample. Starting with the highly informal ideas of Kurdyumov, he eventually produced the paper [2], which presumably contains the first counterexample to the Positive Rates Conjecture. I say "presumably" because I believe there is no one else in the world besides Gacs who understands that paper. When the preprint of it first became available in 1984, I was visiting at UCLA. Rick Durrett and I tried unsuccessfully for about a month to read it. We then invited Gacs to visit us and explain it all. He graciously accepted our invitation, but we still were unable to understand the example. Too many of the details were missing, and too much of the write-up was in a style that was highly unconventional for a mathematical work. Nevertheless, I began to think that there was indeed a rigorous counterexample hidden somewhere.

Experts in the field began to pressure Gacs to write a more accessible version. It has taken almost 15 years. The result is the long paper following this article. The example contained in that paper is much improved over the earlier version, as I will explain in Section 4. Gacs has included all of the details, and they meet the commonly accepted standards for mathematical rigor. It is an extremely difficult work to read (partly because of the presence of all the details!), but I am convinced that this complexity is unavoidable, barring further major breakthroughs.

When the paper was completed, I was asked by Joel Lebowitz to referee it for this journal. I spent over a year sporadically attempting to make my way through it. Eventually, I read most of it and became convinced that it was essentially correct. Nevertheless, I had many questions. So I obtained permission from Joel to reveal my identity as referee to Gacs. During two long weekends together, we worked through the paper line by line. Many mistakes were corrected, and all of my questions were answered. It would be foolhardy for me to guarantee that all of the mistakes have been removed. But I do assert that any remaining mistakes are minor and easily fixed. The result is correct. It is also quite beautiful, in spite of the fact that only the most dedicated reader will be able to get through it. In Section 5, I will explain all of the main ideas that make the example work.

#### 4. FREQUENTLY ASKED QUESTIONS

**What kind of cellular automaton is the Gacs counterexample?** The model has a noise parameter  $\varepsilon$ , and it is deterministic when  $\varepsilon = 0$ . It is (of course) one-dimensional, and it has nearest-neighbor interactions. So in these respects, it is like the examples of Section 2. It differs from those models in that its state space is very large (but finite), and it is not monotone. Gacs could have chosen to use the state space  $\{-1, +1\}$ , at the cost of losing the nearest-neighbor property, but his model has an essential lack of monotonicity that cannot be avoided. If you ignore part of the state space, then you could say that there are ground states. The state space can be roughly thought of as a product of two main components. The first component contains the part of the state that is to be preserved at each site, and the second component is the part that makes this preservation possible. It is the first part that is associated with the ground states. In my simplified version of the model, to be described in Section 5, the part that is preserved is called "Address".

**How big is the state space?** The model depends on some interrelated parameters that can be chosen in various ways. The relationships that must be satisfied by these parameters are found in the first part of Section 9 in Gacs’s paper. A certain constant  $R_0$  is involved that is known to be finite, but unfortunately I don’t know how large it is. Once  $R_0$  is determined, the number of states can be set equal to  $R_0^{4/3}$ . Various considerations involving relationships to other parameters lead me to conclude that  $R_0$  should be at least  $2^{18}$ , but it may need to be much larger. I doubt it is larger than  $2^{300}$ . Thus, the number of states is probably somewhere between  $2^{24}$  and  $2^{400}$ . These numbers may seem huge, but perhaps they should be viewed in a different way. In the language of computers,  $2^{24}$  states is only 3 bytes, and  $2^{400}$  is only 50 bytes. This language is appropriate in view of the way in which Gacs describes his model, namely as a one-dimensional array of small computers, one computer for each site. Each computer needs at most 50 bytes of RAM.

**What error rate can be tolerated by the model?** Gacs’s proof gives an upper bound of  $\varepsilon \leq R_0^{-40}$ . However, no attempt was made in the proof to optimize either this value, or the size of the state space.

**Why is the Gacs paper so long?** The paper is apportioned into five roughly equal chunks: (i) exposition, (ii) definitions and examples involving the most important mathematical issues, (iii) proofs connected with those issues, (iv) definitions and proofs involving the details of the transition function, (v) results that go beyond what is needed to disprove the Positive Rates Conjecture. Of course, these chunks are somewhat intermingled. I don’t see how the first three chunks can be shortened significantly, although we may eventually have a better understanding of the whole problem that will allow for a more compact presentation.

The fourth chunk is needed because of the high degree of skepticism that has been expressed towards this work. In many respects, the details of the transition function are quite arbitrary. There would be many ways to implement the necessary procedures, just as there are many ways to write a computer program to accomplish a particular task. Once the overall picture is well-understood, the details are not so interesting to many people. But some version of those details should appear somewhere, and this is the right place for them. Furthermore, there may be some experts who will find something of value in the details; Gacs used many clever tricks in this part of the paper to try to make things work as smoothly as possible. Indeed, the programming task required in working out the details of the transition function is not a standard one, since it involves “parallel processing” on the sites of the lattice. In contrast to ordinary programs, no universally accepted standard idiom has emerged in computer science for the writing of such programs. Also, proofs of properties of such programs are notoriously complicated.

The fifth chunk is included for two reasons. The first is that it contains results that are interesting in their own right, apart from the Positive Rates Conjecture. The second reason has to do with Gacs’s intellectual honesty: if he was going to publish such a long paper, he wanted it to contain some interesting results that were not already in his 1986 paper [2].

**What are these additional results?** There are at least four significant results in the paper that go beyond what is needed for a counterexample to the Positive Rates Conjecture. The first of these already appeared in the 1986 version, while the others are new. They are: (i) the model is capable of universal computation;

(ii) it can be implemented in continuous time; (iii) the model can be made “self-organizing”; (iv) it has the “positive capacity property”. Result (i) implies that the model is capable of arbitrarily complex behaviors that are stable in the presence of noise. Result (ii) is not surprising, but it is technically quite difficult to prove, due to the fact that the array of “computers” forming Gacs’s model need to work in parallel. The intrinsic lack of synchronization in continuous-time models makes parallel operations difficult. Result (iii) says that the model is capable of highly organized behavior, even if the initial configuration is relatively disorganized. Result (iv) has to do with the ability of the model to store information reliably. The two-dimensional Toom model is capable of reliably storing one bit of information in the presence of noise. This single bit is, namely, whether the initial configuration is “all -1” or “all +1”. The Gacs model is capable of reliably storing one bit of information *per site* in the presence of noise.

**Aside from the results just mentioned, in what other ways does Gacs’s present paper improve on the 1986 paper?** The most important way is in readability. The 1986 paper was written too much like a computer program. The current paper also contains parts that look like a computer program, but those parts involve the specific details of the transition function, for which it is actually necessary, for the sake of the proof, to use a kind of programming language. I discuss the reason for this at the end of 5.3. Another significant difference involves the initial configurations. The configurations corresponding to “ground states” in the current paper are much simpler than the ones that were used in the 1986 version. Also, many of the basic concepts needed in the model are now better understood. As a consequence, parts of the proof have been simplified or made more efficient. Finally, if you care about seeing all the details, you won’t be able to find nearly as many of them in the 1986 paper as you can in the current paper.

**What is the main idea that makes the model work?** Here is a short answer. The model uses a hierarchy of majority vote and contact procedures to correct errors. At the lowest level, these procedures work much like the transition rules for the majority vote and discrete-time contact models of Section 2 to restore order to the system when simple errors occur. Periodically, more complicated majority vote and contact procedures are carried out on successively larger scales, so that eventually, any finite island of errors can be corrected. These higher order operations are made possible by “self-simulation”, which is described in detail in 5.3. All of the other main ideas behind the model can be found in my explanation in Section 5. For the complete details, you will, of course, need to read Gacs’s paper.

**Why is this result so important?** From a mathematical point of view, the answer is that it solves a problem that has puzzled a lot of people for many years, and it does so by introducing several new ideas into the study of cellular automata. Gacs’s cellular automaton behaves in a manner that is completely unfamiliar. The mechanisms that make it work are likely to be useful in the future.

From the point of view of computer science, the importance stems from the fact that Gacs’s model can be thought of as an array of simple computers. Even though I have restricted my discussion to the infinite lattice  $\Lambda = \mathbb{Z}$ , Gacs’s results also have implications for large finite arrays. Essentially, he shows that there exists a simple finite-state machine  $M$  such that for any computer program  $\pi$  and any desired level of reliability  $p < 1$ , the program  $\pi$  can be run with reliability  $p$  on

a sufficiently large one-dimensional array of copies of  $M$ , each one communicating only with its nearest neighbors, even if these machines all have the same small but positive error rate. The size of the array is determined only by the memory and time resources required by the program  $\pi$  and by the reliability level  $p$ . A compiler for such a program would simply translate  $\pi$  into an initial configuration for the array. Such a compiler could be made independent of the size of the array. I would also think that computer scientists would be interested in the details of the hierarchical error-correcting techniques used in Gacs's model.

Gacs's result also has importance in other fields of science. We have already discussed some of the connections to statistical mechanics. In a different direction, I think that the self-organizing feature might be of interest to biologists. Gacs shows in his paper that highly organized arrays can form spontaneously out of relatively chaotic initial conditions, and that these arrays are capable of reliably carrying out complex tasks, even if the individual components in the array are unreliable. One could easily speculate here about possible implications for molecular and evolutionary biology.

**One last time: Is the proof correct?** Yes. Once you understand the principles that make it work, you also understand that the argument is very robust.

## 5. HOW IT WORKS

My description will be in five parts, which I think correspond to the five key ideas that make the model work. I will describe a simplified version of Gacs's model. In many of the details, my version will not agree with his version. There are two reasons for this. First, I am not trying to explain all of the results obtained by Gacs, so I don't need some of the advanced features found in his model. Second, it turns out that some things are best treated one way at the informal level of this article and another way when all of the details are included. The differences have to do with efficiency of presentation, not with essential ideas.

It is my opinion that the version presented here can be fleshed out to become entirely rigorous. Of the five propositions stated below, I consider my arguments in favor of Propositions 1 and 2 to be reasonably complete. My discussion of Proposition 3 is less complete, mainly because it requires some knowledge of ideas from computer science that don't fit into this article. However, the necessary details can be found in a relatively short and self-contained part of Gacs's paper (Sections 7.3 and 7.4). My argument in favor of Proposition 4 is close to being a rigorous proof, except that it relies on Proposition 3. Proposition 5 is the most critical one. I am able to give you the main ideas of the proof, but I must omit many details (which I have worked out for myself). However, I haven't hidden any serious difficulties.

Let's begin. One of the most obvious differences between Gacs's version and mine is that I will use the following range-5 interaction neighborhood:

$$\mathcal{N} = \{-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5\}.$$

Gacs's model has nearest-neighbor interactions. By using this expanded neighborhood, I can make the state space somewhat simpler.

You should imagine that at each site of the lattice  $\Lambda = \mathbb{Z}$ , there is a small computer. In keeping with the terminology in Gacs's paper, I will call such a computer a *cell*. The state of each cell can be described by a finite *bit string*, or in

other words, by a finite sequence of 1's and 0's. Thus, the state space is  $\mathbb{S} = \{0, 1\}^K$  for some positive integer  $K$ .

It will be convenient to think of a state as being composed of several shorter bit strings that have been concatenated together. Each of these shorter strings is called a *field*, and each field will have a name. There are six of them: *Address*, *Age*, *Flags*, *SimBit*, *Workspace*, *Mailbox*.

The first two fields form what I call the *local structure*, while the last three fields are the *simulation structure*. The Flags field controls error-correction in the local structure, and it also provides the means by which the simulation structure influences the local structure.

For each of the local structure fields, I will think of the bit string in that field as a binary representation of a positive integer. So, for example, if I were to say that a certain field of five bits has value 5, then I would mean that the five bits take the values 00101. And if I say a field takes the value 0, then I mean that all of the bits in the field equal 0.

The value of the Address field is an integer between 0 and  $Q - 1$ , where  $Q$  is a parameter called the *colony size*. In the initial configuration, the Address field at  $x$  has value  $x \pmod{Q}$ , and one of the main goals of the model is to preserve these initial values, and to restore them where errors occur. When the initial values are preserved, the cells are naturally organized into *colonies* consisting of  $Q$  cells, with the Address field indicating the location of a cell within the colony. The significance of these colonies will be explained later, particularly in 5.2 and 5.3. It is natural to let  $Q$  be a power of 2, so that the Address field contains  $\log_2 Q$  bits. In what follows, I will assume that  $Q \geq 2^{13}$ .

The value of the Age field is an integer between 0 and  $U - 1$ , where  $U$  is another parameter, called the *work period*. In my version of the model, I let  $U = 128Q$ , so the Age field contains  $7 + \log_2 Q$  bits. The cells will repeatedly go through a cycle of transitions, with each cycle containing  $U$  time steps. In the initial configuration, the Age field has value 0 at all sites. Under normal circumstances, the value of the Age will change by  $+1 \pmod{U}$  at each time step.

The Flags field contains two bits, which I call Flag1 and Flag2. Roughly speaking, if Flag1 has the value 0, then a cell is operating normally. A value of 1 for Flag1 indicates that there is some sort of emergency, presumably caused by transition errors. In such a case, the value of Flag2 controls the "direction" of recovery operations. The initial value for both Flag1 and Flag2 is 0 at all sites. The transition rules for the Flags field are given in 5.2. Flags is a simplified version of the "Kind" field in Gacs's paper.

For the simulation structure fields, I will think of the bit strings as representing information. The Workspace field is the place for storing information in a cell, and the Mailbox field is the place for exchanging information among several cells. I will not give the sizes of these fields explicitly. However, it can be shown that they are bounded above by  $a \log_2(QU)$ , where  $a$  is a constant that does not depend on  $Q$  or  $U$ . In the initial configuration, all of the bits of the Mailbox and Workspace fields are set equal to 0.

The SimBit field contains five bits. The first of these bits is called the *primary SimBit*. By combining the primary SimBits of a colony, a bit string is obtained that is used to represent the state of a single "simulated cell". I will explain in 5.3 all about the simulation that is to be carried out by the system. In 5.4, I will give

the initial values for the SimBit fields. The remaining four bits in the SimBit field are used for error-correction, as described in 5.4.

The phase transition will come from the Address field. The initial values of this field are periodic in the lattice, with period  $Q$ . The elaborate error-correcting properties of the model, to be described in the remainder of this section, ensure that if  $\varepsilon > 0$  is sufficiently small, then for any given site  $x$  and time  $t$ , the value of Address at  $x$  at time  $t$  will equal its initial value with high probability. Assuming that this claim is true, a standard argument shows that there must be a stationary measure for the system that puts high probability on the event that at any given site  $x$ , the Address field equals  $x \pmod{Q}$ . Shift this stationary measure along the lattice to obtain  $Q - 1$  other stationary measures. Thus, if we restrict our attention to the Address field, the system will have  $Q$  different stable ground states, and hence a phase transition.

There is another way to view the stability of the Address field values. One could say that the system has the ability to “remember” the initial values of the Address fields, which is to say that it can reliably store  $\log_2 Q$  bits of information. Gacs’s more sophisticated model has the ability to reliably store an infinite amount of information. More precisely, it can store one bit per site. That is, a one-bit field can be included in the state of each cell, with the initial value being chosen arbitrarily at each site, and the system will maintain that part of the initial configuration with high probability for all time. This gives Gacs’s model uncountably many different stationary measures.

**5.1. The structure of the random error set.** My goal in this subsection is to take care of the probabilistic part of the argument up front, so that the rest of the discussion can focus as much as possible on the properties of the deterministic system. Recall that there are two sources of randomness: the random error set  $E$  and the noise variables  $s(x, t)$ . The model will be so robust, that the values taken by the noise variables will be irrelevant, even if there is a “malicious demon” choosing the values of  $s(x, t)$ .

I cannot be so cavalier about the error set  $E$ . In this subsection, I will decompose  $E$  into pieces called “level- $n$  errors”, where  $n \geq 0$ . These pieces are certain types of clusters of errors in  $E$ . It will be shown that they are extremely sparse for large  $n$ . In 5.2 and 5.4, I will explain how the system is able to correct all level-0 and level-1 errors, and in 5.5, I will explain how the higher-level errors are dealt with.

In defining various types of clusters of errors, it will be useful to have some simple terminology for the degree to which two sets are separated in space-time. Let  $A, B$  be subsets of the space-time set  $\mathbb{Z} \times \{0, 1, 2, \dots\}$ . For positive integers  $m, n$  I will say that  $A$  and  $B$  are  $(m, n)$ -linked if there is a space-time box of the form  $[x, x + m) \times [y, y + n)$  that contains at least one member of each of the sets  $A$  and  $B$ . If  $A$  and  $B$  are not  $(m, n)$ -linked, then I will say that they are  $(m, n)$ -separated.

Let  $E$  be the random error set. A *candidate level-0 error* is a nonempty subset of  $E$  that is contained inside a space-time box of the form  $[x, x + 1] \times [t, t]$ . In other words, it is either a set consisting of a single error, or it is a set containing two errors at adjacent sites. A candidate level-0 error  $S$  is a *level-0 error* if the two sets  $S$  and  $E \setminus S$  are  $(24, 24)$ -separated. This condition means that  $S$  is required to be isolated from the rest of  $E$  by a certain amount. The number 24 is simply a convenient choice that makes later estimates work out well. The union of all level-0 errors  $E$  is called *level-0 noise*. It is denoted by  $E_0$ .

Now I continue inductively. Fix  $n > 0$  and suppose that I have defined candidate level- $k$  errors, level- $k$  errors, and the level- $k$  noise  $E_k$  for  $k < n$ . A nonempty set  $S \subseteq (E \setminus E_{n-1})$  is a *candidate level- $n$  error* if (i)  $S$  is contained in a  $Q^n \times U^n$  space-time box, and (ii)  $S$  contains at least two disjoint candidate level- $(n-1)$  errors that are  $(24Q^{n-1}, 24U^{n-1})$ -linked. A candidate level- $n$  error  $S$  is a *level- $n$  error* if (iii)  $S$  does not contain two candidate level- $n$  errors that are  $(104Q^{n-1}, 104U^{n-1})$  separated, and (iv) the sets  $S$  and  $E \setminus (S \cup E_{n-1})$  are  $(24Q^n, 24U^n)$ -separated. *Level- $n$  noise*, denoted by  $E_n$ , is the union of  $E_{n-1}$  and all of the level- $n$  errors in  $E$ .

Informally, the definitions say the following: a candidate level- $n$  error  $S$  in  $E$  must be (i) small enough to fit into a space-time box of a certain size, and (ii) dense enough to contain at least two distinct candidate level- $(n-1)$  errors that are not too far apart. A candidate level- $n$  error  $S$  qualifies as a real level- $n$  error if (iii)  $S$  is not big enough to contain two smaller candidate level- $n$  errors that are separated from each other by a certain amount, and (iv)  $S$  is sufficiently isolated from the rest of  $E \setminus E_{n-1}$ . Note that if condition (iii) fails to hold for a candidate level- $n$  error  $S$ , then  $S$  qualifies as a candidate level- $(n+1)$  error.

In conjunction with these definitions, I will need some facts about  $E$ . The first is that, for  $n > 0$ , every point in  $E \setminus E_{n-1}$  belongs to a candidate level- $n$  error  $S$ . I can prove this by induction on  $n$ . As is often the case with such proofs, things are made easier if I prove a little more. A candidate level- $n$  error  $S$  is called *minimal* if  $S$  contains precisely  $2^n$  points (it is easy to see that a candidate level- $n$  error cannot contain less than  $2^n$  points). I will show inductively that every point in  $E \setminus E_{n-1}$  belongs to at least one minimal candidate level- $n$  error  $S$ . The case  $n = 1$  is an easy exercise that is left for the reader.

For the inductive step, assume that there exists a space-time point  $(x_1, t_1)$  that is a member of  $E \setminus E_{n-1}$  for some  $n > 1$ . By the inductive hypothesis,  $(x_1, t_1)$  belongs to a minimal candidate level- $(n-1)$  error  $S_1$ . The minimality implies that  $S_1$  consists of two disjoint minimal candidate level- $(n-2)$  errors that are  $(24Q^{n-2}, 24U^{n-2})$ -linked. It follows that  $S_1$  fits into a  $26Q^{n-2} \times 26U^{n-2}$  space-time box. It is easy to see from the definition that if any point in  $S_1$  is part of a real level- $(n-1)$  error, then  $S_1$  itself would be contained in a real level- $(n-1)$  error. But  $S_1$  contains  $(x_1, t_1)$  and  $(x_1, t_1) \notin E_{n-1}$ , so  $S_1$  cannot be contained in a real level- $(n-1)$  error. Since  $S_1$  clearly satisfies condition (iii) for level- $(n-1)$  errors, it follows that  $S_1$  fails to satisfy condition (iv). Thus, there exists a point  $(x_2, t_2) \in E \setminus E_{n-1}$  such that  $(x_2, t_2) \notin S_1$ , and such that  $(x_2, t_2)$  and  $S_1$  are  $(24Q^{n-1}, 24U^{n-1})$ -linked. For reasons that will become clear in the next paragraph, we will consider a slightly different collection of choices for this point, namely all points  $(x_2, t_2) \in E \setminus E_{n-1}$  such that  $(x_2, t_2) \notin S_1$  and such that  $(x_2, t_2)$  and  $(x_1, t_1)$  are  $(25Q^{n-1}, 25U^{n-1})$ -linked. Since I assumed earlier that  $Q \geq 2^{13}$  and  $U = 128Q$ , and since  $S_1$  fits inside a  $26Q^{n-2} \times 26U^{n-2}$  space-time box, this increases the set of possible choices for the point  $(x_2, t_2)$ . Apply the inductive hypothesis to such a point  $(x_2, t_2)$  to find a minimal candidate level- $(n-1)$  error  $S_2$  containing  $(x_2, t_2)$ . As with  $S_1$ , we know that  $S_2$  fits into a  $26Q^{n-2} \times 26U^{n-2}$  space-time box. Let  $S = S_1 \cup S_2$ .

I claim that  $S_2$  can be chosen so that  $S_1$  and  $S_2$  are disjoint. It is easy to see that if this claim is true, then the set  $S$  is a minimal candidate level- $n$  error containing  $(x_1, t_1)$ , and we are done. In order to prove the claim, we just need to choose the point  $(x_2, t_2)$  so that it is far enough away from  $(x_1, t_1)$ . If the set of possible choices

for  $(x_2, t_2)$  contains a point that is  $(52Q^{n-2}, 52U^{n-2})$  separated from  $(x_1, t_1)$ , then we let  $(x_2, t_2)$  be that point, and the size restrictions on  $S_1$  and  $S_2$  force them to be disjoint. Otherwise,  $S$  fits inside a  $104Q^{n-2} \times 104U^{n-2}$  space-time box, and  $S$  is  $(25Q^{n-1} - 104Q^{n-2}, 25U^{n-1} - 104U^{n-2})$ -separated from the rest of  $E \setminus E_{n-1}$ . Under my assumption about the sizes of  $Q$  and  $U$ , this would imply that  $S$  fits into a  $Q^{n-1} \times U^{n-1}$  space-time box and is  $(24Q^{n-1}, 24U^{n-1})$ -separated from the rest of  $E \setminus E_{n-1}$ . Thus  $S$  would satisfy conditions (i) and (iv) in the definition of level- $(n-1)$  errors. Clearly,  $S$  satisfies condition (iii), and since  $S$  contains  $S_1$ , it also satisfies condition (ii). So  $S$  would be a real level- $(n-1)$  error. As explained earlier,  $S_1$  cannot be part of a real level- $(n-1)$  error. This leaves us with the first possibility, which is that  $(x_2, t_2)$  can be chosen to be  $(52Q^{n-2}, 52U^{n-2})$ -separated from  $(x_1, t_1)$ , thereby giving us the desired disjointness and completing the inductive proof.

The argument just given has an interesting consequence about the size of a level- $n$  error. First of all, after replacing  $n-1$  by  $n$ , the argument shows that a minimal candidate level- $n$  error  $S_1$  fits into a  $26Q^{n-1} \times 26U^{n-1}$  space-time box. It further shows that if there is a point in  $E \setminus E_{n-1}$  that is  $(52Q^{n-1}, 52U^{n-1})$ -separated from at least one member of  $S_1$ , then that point lies in a second minimal candidate level- $n$  error  $S_2$  that is disjoint from  $S_1$ , and  $S_2$  also fits into a  $26Q^{n-1} \times 26U^{n-1}$  space-time box. If it also happens that  $S_1$  and  $S_2$  are  $(104Q^{n-1}, 104U^{n-1})$ -separated but  $(24Q^n, 24U^n)$ -linked, then conditions (iii) and (iv) prevent  $S_1$  (and  $S_2$ ) from being a part of a real level- $n$  error. Thus, if  $S_1$  is part of a real level- $n$  error  $S$ , any point in  $S \setminus S_1$  must be  $(130Q^{n-1}, 130U^{n-1})$ -linked with  $S_1$ . We may therefore conclude that every level- $n$  error  $S$  actually fits easily into a  $200Q^{n-1} \times 200U^{n-1}$  space-time box. In particular, a level-1 error fits into a  $200 \times 200$  space-time box.

I need one more fact about level- $n$  errors. For  $n \geq 0$ , let  $\varepsilon_n$  denote the *level- $n$  error rate*, which is the probability that a given space-time box  $B$  of size  $Q^n \times U^n$  has nonempty intersection with some candidate level- $n$  error. Clearly  $\varepsilon_0 = \varepsilon$ . To obtain an upper bound on  $\varepsilon_n$ , we first consider the event that a given  $Q^n \times U^n$  space-time box has nonempty intersection with at least one candidate level- $(n-1)$  error. Clearly, the probability of this event is bounded above by  $QU\varepsilon_{n-1}$ . Since a candidate level- $n$  error must contain at least two disjoint candidate level- $(n-1)$  errors, the Kesten-Vandenberg Inequality (see [1]) implies that  $\varepsilon_n \leq (QU\varepsilon_{n-1})^2$ . As a consequence,

$$\varepsilon_n \leq (QU)^{2^{n+1}-2} \varepsilon^{2^n} < (Q^2U^2\varepsilon)^{2^n} \text{ for } n > 0.$$

Thus, when  $\varepsilon < (QU)^{-2}$ , the candidate level- $n$  errors in  $E$  become extremely rare as  $n$  increases.

Based on these facts, a simply Borel-Cantelli argument shows that, with probability 1, any given member of  $E$  belongs to a candidate level- $n$  error for only finitely many values of  $n$ , and so  $E_n \nearrow E$  as  $n \nearrow \infty$ , as long as  $\varepsilon$  is sufficiently small. I will summarize these results as:

**Proposition 1.** *For  $n > 0$ , every level- $n$  error fits into a  $200Q^{n-1} \times 200U^{n-1}$  space-time box, and if  $\varepsilon_n$  is the level- $n$  error rate defined above, then*

$$\varepsilon_n \leq (Q^2U^2\varepsilon)^{2^n}.$$

*If  $\varepsilon < (QU)^{-2}$ , then with probability 1, every error belongs to a level- $n$  error for some  $n \geq 0$ .*

The current version of Gacs's construction does not contain a similar decomposition of the error set explicitly. What corresponds to it is the somewhat more sophisticated notion of a "robust medium", and the "simulation damage probability bound" found in Gacs's Lemma 8.4. But for our purposes here, the concept of level- $n$  noise works just fine.

**5.2. The dynamics of the local structure and Flags fields.** The transition rules for the local structure fields depend on the value of the Flags field, so it is best if I give the transition rules for Flags first. In order to do that, I will need some more notation and terminology. Given a site  $x$ , let

$$R(x) = \{x + 1, x + 2, x + 3, x + 4, x + 5\}$$

$$L(x) = \{x - 1, x - 2, x - 3, x - 4, x - 5\}.$$

Given a configuration  $\xi$ , I will now define the *apparent colony* of a cell at  $x$ , to be denoted by  $C(x, \xi)$ , or more frequently, by  $C(x)$  when the configuration  $\xi$  is clear from the context. If at least three of the sites in  $R(x)$  have Addresses that are consistent with each other, then extrapolate from those Address values to obtain an interval of  $Q$  cells that contains  $x$  and is consistent with the three Addresses in  $R(x)$ , and call this interval the apparent colony  $C(x)$ . If three such sites do not exist in  $R(x)$ , then  $C(x)$  does not exist. Here are three examples to help clarify this definition. If the Addresses in  $R(x)$  are 0, 1, 29, 3, 81, then  $C(x)$  exists because the Address values at  $x + 1, x + 2, x + 4$  are compatible with each other. In this case,  $C(x)$  consists of the sites  $x - Q + 1, \dots, x$ . If the Addresses in  $R(x)$  are 13, 14, 15, 16, 2, then  $C(x)$  is  $x - 12, \dots, x - 13 + Q$ . And if the Addresses in  $R(x)$  are 3, 4, 6, 5, 2, then  $C(x)$  does not exist, because no three of these sites have compatible Address values.

I will say that there is *inconsistency* at  $x$  if one of the following four conditions holds: (i)  $C(x)$  does not exist, or (ii)  $C(x)$  exists, but there are at least three sites in  $L(x) \cap C(x)$  whose Address values are not consistent with the positions of those sites relative to  $C(x)$ , or (iii) there do not exist three sites in  $R(x)$  with the same Age value, or (iv) there exist at least three sites in  $R(x)$  with the same Age value  $a$  and at least three sites in  $L(x) \cap C(x)$  whose Age value differs from  $a$ . Otherwise, there is *consistency* at  $x$ . The main purpose of the transition rules in this section is to restore consistency when damage is done to the system by level-1 noise. This purpose is accomplished through a combination of majority vote procedures involving the Address and Age fields. The Flag1 bit helps control these procedures. The Flag2 bit is merely a nuisance in this part of the discussion, but I include it for the sake of completeness. I strongly recommend that you ignore any mention of the Flag2 bit at this stage. It will only become significant in 5.5.

The transition rules for Flags depend on the local structure fields, and also on two particular bits in the Workspace field. I call these two bits Workspace.Flag1 and Workspace.Flag2. The transition rules for these two bits will be given in 5.5. The circumstances are quite rare under which these two bits have any influence on the Flags field, so it is best not to worry about them on first reading. In particular, you may safely ignore condition (iii) in the following until you get to 5.5.

Here are the transition rules for Flag1:

$0 \rightarrow 1$  if at least one of the following three conditions is satisfied: (i) there is an inconsistency at  $x$ , or (ii) at least three sites in  $R(x) \cap C(x)$  have Flag1 equal to 1, or (iii) at least

three sites in  $N(x) \cap C(x)$  have `Workspace.Flag1` equal to 1

$1 \rightarrow 0$  if conditions (i) and (iii) do not hold and the following condition does hold: (a) no more than one site in  $R(x) \cap C(x)$  has `Flag1` equal to 1.

In all other cases, the value of `Flag1` does not change. Conditions (i) and (ii) imply that inconsistencies cause 1's, and that a block of three or more 1's grows towards the left at speed 3, until it reaches a site whose apparent colony is disjoint from the block. We will see that this part of the rule is important in one of the early stages of recovery from level-1 errors. Condition (iii) shows that the `Flag1` field can get its value from the `Workspace` field, but this condition does not play any significant role until 5.5. Condition (a) shows how the `Flag1` bit returns to 0 in a "healthy" colony. This rule takes care of isolated errors in the `Flag1` field, such as occur in level-0 noise, and it also is important in the final stage of recovery from level-1 errors, as described below.

I define the *computed value* of `Flag1` at  $x$  to be the value given by the rules in the preceding paragraph. This value plays a role in the transition rules for `Flag2`, which are given here. It is best to ignore these rules until you get to 5.5.

$0 \rightarrow 1$  if at least one of the following four conditions is satisfied: (i) at least four sites in  $L(x) \cap C(x)$  have `Flag2` equal to 1, or (ii) the computed value of `Flag1` at  $x$  is 1 and at least four sites in  $L(x)$  have `Flag2` equal to 1, or (iii)  $C(x)$  does not exist and the computed value of `Age` at  $x$  is divisible by 16, or (iv) at least three sites in  $N(x) \cap C(x)$  have `Workspace.Flag2` equal to 1

$1 \rightarrow 0$  if conditions (iii) and (iv) do not hold and at least one of the following two conditions does hold: (a) the computed value of `Flag1` at  $x$  is 0 and no site in  $L(x) \cap C(x)$  has `Flag2` equal to 0, or (b) the computed value of `Flag1` at  $x$  is 1 and no site in  $L(x)$  has `Flag2` equal to 1.

The *computed value* of `Flag2` at  $x$  is the value given by these rules. Conditions (i) and (ii) imply that a block of 1's in the `Flag2` field will spread to the right at speed 2, but such a wave can only enter a new colony if the `Flag1` values allow it. Thus, the expansion is normally restricted within colony boundaries. Conditions (iii) and (iv) are important in 5.5. Condition (a) shows how 1's in the `Flag2` bits return to 0 within a "healthy" colony. Condition (b) is mostly intended for "unhealthy" colonies.

The transition rules for `Address` at a site  $x$  require majority votes of the `Address` values at certain sites in  $N(x)$ . For such votes, I follow the convention that if the `Address` field at some site  $y$  is involved in a majority vote that is being used to determine the new `Address` value at a site  $x$ , then the `Address` value at  $y$  needs to be adjusted by adding  $x - y \pmod{Q}$  before using it in the vote, to compensate for the position of  $y$  relative to  $x$ . The transition rules for `Age` at a site  $x$  require majority votes of the `Age` values at certain sites in  $N(x)$ , but no adjustment is needed before such votes are taken. However, whenever a majority `Age` value is determined, I follow the convention that the result must be incremented by  $+1 \pmod{U}$  before using it as the new value of the `Age` at  $x$ . In all cases, if the voting

does not produce a clear majority when a transition at  $x$  is being determined, then my convention is that outcome of the majority vote is defined to be the current value at  $x$ . With these conventions in mind, it is now possible to give identical statements for the transition rules of Address and Age. Even though Flag2 plays a role here, it is safe to ignore it until 5.5.

If  $C(x)$  exists and either the computed value of Flag1 at  $x$  is 0 or the computed value of Flag2 at  $x$  is 1, take the majority vote in  $R(x)$ . (Note that, for the Address field, this will give a value that matches the position of  $x$  in  $C(x)$ .)

Otherwise, take the majority vote in  $L(x)$ .

The *computed values* of Address and Age are the ones determined by these rules. For errors that arise in the local structure or Flags fields due to level-0 noise, it does not matter which of these two rules is in effect. Such errors will be corrected immediately. In fact, a simpler type of majority vote could have been used if we were only worried about level-0 noise. The slightly complicated nature of the rules is necessary to handle level-1 noise efficiently, as I will now explain.

Let  $S$  be a level-1 error. Recall that  $S$  must be contained in a  $200 \times 200$  space-time box. This box can touch at most two colonies. For simplicity, let us suppose that it touches exactly two colonies, which I call the “damaged colonies”. By definition, all other level-1 errors are at least  $(24Q, 24U)$ -separated from  $S$ . Since I will show that the damage caused by  $S$  is corrected within a space-time box of size  $2Q \times 2U$ , we can assume that the noise is restricted to level-0 outside of  $S$ . Under this assumption, the computed values of all of the local structure fields are the same as in the deterministic system, except in those cells that are influenced by the damage in  $S$ . At any given time, I define the “damage island” to be the smallest interval of cells whose computed Address and Age values don't equal the values that they have at that same time in the deterministic system. My main goal is to show that the damage island disappears. I will also show that the Flags fields are returned to 0.

For the reader that is not interested in lots of detail at this stage, here is the main idea. If there are any incorrect values of Address or Age in the damage island, the resulting inconsistencies will cause the Flag1 bits in the island to change to 1, due to conditions (i) and (ii) in the transition rules for those bits. Once this happens, the local structure fields in the island are determined by majority voting from the left, so that correct values of those fields flow into the island from the undamaged colony on the left. The damage cannot expand to affect the cells of the undamaged colony on the right, for the following reasons: (1) the apparent colony of those cells equals the real colony, so that (2) no inconsistencies arise in the undamaged colony, implying that (3) the Flag1 bits in the undamaged colony remain equal to 0, further implying that (4) their local structure fields are determined by majority voting from the right. Since the island shrinks at a steady rate on the left, and since its growth on the right is bounded, the island soon disappears, as desired. If you can accept this informal argument, you may want to skip ahead to the statement of Proposition 2.

Throughout the following, I will talk about various “waves” in describing the way in which certain values spread throughout the damaged colonies. For example, a block of three 1's in the Flag1 bit can initiate a leftward-moving wave of 1's,

due to condition (ii) in the rules for Flag1. It is easy to check that the front of this particular type of wave moves at speed 3 in the absence of noise. Other types of waves have their own characteristic speeds, and these speeds are important in the argument. Since I need to worry about the influence of level-0 noise, I want to make a general comment on how such noise affects the speeds of the various types of waves. When a level-0 error interacts with the front of a wave, it can slow it down a little. By definition, each level-0 error is restricted to at most two neighboring sites, and all level-0 errors are at least  $(24, 24)$ -separated from one another. Because of this sparsity, routine checking in each case shows that the speed of a wave is reduced on the average by much less than  $1/4$  cell per time step. I will not bother to give the details here. In the future, I will sometimes refer to this kind of speed reduction as a “transient effect”.

Let me first show that the damage island cannot expand towards the left, except for occasional temporary expansions due to level-0 noise or the Flag2 bit. At the left end of the damage island, an inconsistency must exist. This inconsistency initiates a leftward-moving wave of 1’s in the Flag1 field. This wave moves at least at speed 2.75 in the presence of level-0 noise. If we could ignore the influence of the Flag2 bit, then we could conclude that the Address and Age fields in the cells to the left of the damage island remain unaffected by the damage in the island, since the majority voting for those cells will come from the left. In the next paragraph, I am going to explain why the Flag2 bit does not change this picture significantly, except to possibly reduce the various wave speeds by another  $1/4$  cell per time step. I refer to this reduction as another “transient effect”. You may ignore the next paragraph until you get to 5.5.

Initially, the Flag2 bits equal 0 in the cells that lie left of the damage island, and those bits can only change to 1 due to conditions (iii) or (iv) in the transition rules for Flag2. Condition (iii) concerns the influence of `Workspace.Flag2`, and the transition rules for `Workspace.Flag2` in 5.5 make it impossible for that bit to have any influence in cells to the left of the island when the noise is restricted to level-0, so I can safely ignore condition (iii). Condition (iv) can only apply once every 16 time steps in the cells to the left of the island. When this happens, only three cells can be affected, and during the next three time steps, the 1’s change back to 0’s and the local structure fields are repaired in those three cells. This argument shows that the computed values of the Flag2 bits remain 0 in all of the cells to the left of the island during an overwhelming majority of the time steps. It follows that the damage island cannot expand leftward in any significant manner. Furthermore, as the left end of the damage island erodes towards the right, the 1’s in the Flag2 bit also get eroded away by the 0’s to the left of the island. The 0’s advance at speed 1, except when delayed by transient effects, so we may assume that the average speed is at least  $1/2$ .

It is a routine matter to check from the various transition rules that in level-0 noise, the computed values of the Flag1 bits must remain equal to 0 in the nearest undamaged colony on the right. The reason is that  $C(x)$  coincides precisely with the undamaged colony itself for all  $x$  inside that colony, thereby ensuring consistency at all such cells, even in the presence of level-0 noise. This fact prevents the damage island from expanding rightward beyond the left boundary of this colony. Therefore, I can restrict my attention to the damaged colonies for the rest of the argument.

Since repairs cannot begin in earnest until after the last error in  $S$  occurs, I will also restrict my attention to the time interval after that error.

Consider the leftmost cell in the damage island. Except for one special situation that will be described below, there must be an inconsistency at this cell, causing the computed value of Flag1 to equal 1 there. (As explained earlier, we can initially assume that the computed Flag2 bits are all 0 to the left of the island during most time steps, from which it follows that the computed value of Flag2 is usually 0 in the leftmost cell in the island.) As a result, the local structure fields in this cell are determined by a majority vote from the left, causing them to change so that they are consistent with the local structure fields outside the island. (Simultaneously, the Flag2 bit becomes equal to its computed value, which is 0.) At least one cell is repaired in this fashion during each time step, except during delays caused by the transient effects. Taking such delays into account, it is still true that the average repair rate is at least  $1/2$  cell per time step. To put it another way, the left end of the island “erodes” at a speed that is at least  $1/2$  (and this erosion includes changing 1’s to 0’s in the Flag2 bit).

The one exception occurs when the leftmost cell in the island happens to be near the left end of its own apparent colony. In this case, it is possible for the computed value of Flag1 bit to equal 0 in this cell, at least initially. However, a little thought reveals that as long as the island covers much less than  $Q$  cells, there must be an inconsistency somewhere within this apparent colony. The nearest such inconsistency produces 1’s that move leftward, and those 1’s eventually reach the left end of the island. Similar waves of 1’s are produced at every inconsistency in the island. Since these waves move at a speed that is slightly less than 3 in the presence of level-0 noise, it is not hard to see that within a short period of time after the end of  $S$ , the computed value of Flag1 equals 1 in every cell in the island that lies near the left end of its own apparent colony. To estimate the length of this “short period”, I note that during the time interval when  $S$  occurs, the damage island can expand at most 800 cells to the right, even with the help of level-0 noise (without noise, the damage can spread at speed 3, so the speed is less than 4 with the noise). Thus, no leftward-moving wave of 1’s needs to cover more than 1000 cells. This fact limits the “short period” to less than 500 time steps, during which the damage island might expand another 2000 cells to the right. The inconsistency at the right end of the island ensures that these extra 2000 cells also have Flag1 bits with computed value equal to 1. Note that since  $Q \geq 2^{13}$ , the damage island has size less than  $Q/2$  at the end of the 500 time steps.

After the 500 time steps, the erosion of the left end of the island proceeds with an average speed that is at least  $1/2$  cell per time step. Since the damage island was originally restricted to size 200, and since the left end has been steadily eroding to the right, it takes at most 400 time steps before the local structure fields of the left damaged colony are completely repaired. From that point on, the damage island is strictly contained in the damaged colony on the right. Also, the island can not have grown more than 1600 cells at the right end during these 400 steps. So the size of the island has remained below  $3Q/4$ . It is easy to see that it takes at least  $Q - 10$  cells to form a block with no internal inconsistencies, so the argument given earlier shows that the left end of the island continues to erode at speed  $1/2$ . At most  $200 + Q$  cells can be damaged during the entire recovery operation, so the island gets completely eroded within  $200 + 500 + 2(Q + 200)$  time steps. Since I

have assumed that  $Q \geq 2^{13}$  and  $U = 128Q$ , this is much less than  $U/32$  time steps. All of the damage to the local structure fields has been repaired. With consistency restored throughout the damaged colonies, the Flag1 bits start changing back to 0's in a wave that moves from the right at a speed that is at least 1.75 in the presence of level-0 noise.

Because the above recovery operation is primarily concerned with the local structure and Flags fields, it is possible that some residual errors remain in the simulation structure fields that could cause delayed effects in the Flags fields. But in the absence of further level-1 or higher errors, such effects will not have any influence on the computed values of the local structure fields. I will show in 5.4 that the simulation structure fields in damaged colonies cannot be affected by a level-1 error in any significant way for more than one colony work period, and that there is no effect at all outside the damaged colonies. Furthermore, the transition rules for `Workspace.Flag1` and `Workspace.Flag2`, to be given in 5.5, only allow those bits to equal 1 during a very specific time interval of length  $2Q$  in the colony work period. Because of this, it turns out that the residual effects on the Flags field are restricted to at most  $U$  time units after the first error in  $S$ . The following proposition summarizes the results about level-1 error-correction in the local structure fields:

**Proposition 2.** *Suppose the errors are restricted to level-1 noise. Then for each level-1 error, there exists a space-time box of the form  $[jQ, (j+2)Q) \times [t, t+U/32)$ , where  $j$  is an integer and  $t$  is a time, such that for all points outside of the union of these boxes and the set  $E_0$ , the local structure fields have the same values that they have in the deterministic system. The same can be said for the Flags fields if the space-time box is enlarged to  $[jQ, (j+2)Q) \times [t, t+U)$ .*

Sections 11, 12, and 15 of Gacs's paper contain his version of the local structure transition rules. Most of the verifications for these rules, which constitute the analogue of Proposition 2, are found in Sections 13 and 14.

**5.3. Self-simulation.** As I will explain in more detail in 5.5, the procedures described in 5.2 cannot fix all of the disruptions that can occur with level-2 and higher errors. Another idea is needed. The germ of this idea goes back to Kurdyumov, but a lot of work was needed by Gacs to actually implement it. It is briefly this: the system will have the capability to simulate itself. The cells are organized into colonies of size  $Q$ . Each colony will play the role of a single *level-1 cell* in a simulated *level-1 cellular automaton*. The state of a simulated level-1 cell will be contained, bit by bit, in the `SimBit` fields of the cells of the colony. At times that are multiples of the work period  $U$ , the simulated states will change. It will be arranged so that these changes follow the same transition rules that exist for the original system. The resulting *self-simulation* will have an important consequence. As I will explain in 5.4 and 5.5, level- $n$  noise will appear as level- $(n-1)$  noise to the simulated system. Furthermore, since the simulated system has the same transition rules as the original, it will also be self-simulating, thereby giving us a *level-2 cellular automaton*. This level-2 system will see level- $n$  noise as level- $(n-2)$  noise. Continuing in this fashion, we obtain a *simulation hierarchy* of increasingly reliable systems. The higher-level systems will be able to help the lower-level systems correct large errors.

To completely describe how all of this works, it will take me the rest of this article. In this subsection, I only want to explain how self-simulation is possible

in cellular automata. I will first give a detailed description of a nontrivial cellular automaton that simulates itself, and then I will give a less detailed description of the way in which Gacs implements self-simulation in his model.

Self-simulation is trickier than it might sound at first. If you try to define self-simulation procedures from the bottom up, you easily get into a vicious cycle. For example, suppose you find that you need to add a procedure at level-0 to make the self-simulation work. Then, of course, you now have more to simulate at level-1. This may mean that you need to enlarge the colony size, which in turn gives you even more to simulate. Fortunately, it all works out.

The following explanation of self-simulation is partly based on a document that was sent to me by Gacs. This is the one part of this introductory paper that I felt uneasy about doing all by myself.

First, let's just talk generally about how to implement a simulation that is not necessarily a self-simulation. Imagine that we have a deterministic cellular automaton  $M_0$  that is supposed to simulate another deterministic cellular automaton  $M_1$ . The state of a cell in  $M_0$  consists of only four fields: two one-bit fields called SimBit and ProgramBit, and two other fields called Workspace and Mailbox. The automaton  $M_0$  is subdivided into colonies, and each colony of  $M_0$  simulates a single cell in  $M_1$ . To find the state of a simulated cell in  $M_1$ , simply read off the bit string obtained by putting all of the SimBits of the corresponding colony together in one string. The initial values of the SimBits are of course determined by the initial configuration of  $M_1$ .

Taken together, the ProgramBits in a colony form a bit string, just like the SimBits. This bit string is simply a "look-up table" for the transition rule of  $M_1$ . If the state of a cell in  $M_1$  is given by a bit string of length  $n$ , and if  $M_1$  has range-5 interactions like the model I have been describing, then such a look-up table will need  $12n2^{11n}$  bits. This number is computed as follows: Each "entry" in the table requires  $12n$  bits, in order to list twelve states: eleven states for the input to the transition function and one state for the output. And there are  $2^{11n}$  entries, corresponding to all possible inputs. So a colony in  $M_0$  will need  $12n2^{11n}$  sites to accommodate this look-up table.

Note that only  $n$  sites are needed to represent the state of a cell in  $M_1$ , so most of the SimBit fields in a colony are not needed to represent the states of the cells in  $M_1$ . This extra space can be used, for example, to contain some special bit string that marks the boundaries of colonies in a way that is independent of the colony size.

During each work period of a colony, three things must happen. First, the colony must determine the states that are represented by it and its ten neighboring colonies. It does this by gathering the relevant SimBits from itself and its ten neighboring colonies, using the Mailbox fields, and then distributing those bits into the Workspace fields of a predetermined set of  $11n$  sites, at the rate of one bit per site. Then these bits get shifted one spatial unit at a time, until they match an entry in the look-up table. Once the match has been found, the new value for the simulated state is taken from the look-up table and moved into the correct SimBit fields.

I imagine that many readers are unfamiliar with the manner in which such procedures are actually implemented, so I will now give some extra detail about how one might organize one of the procedures, namely the first one, involving the

Mailbox fields. First, we need to get the state being simulated by a colony into the Mailbox fields. I need to do this in a way that does not depend on the size of the colony, and yet is properly synchronized. In particular, I want all of the colonies to do this at the same time, and I want the Mailbox field to have a fixed size, no matter what I am simulating. To synchronize things, I can use a small subfield of the Workspace that I could call Signal. At any given time, the Signal field is nonzero in only one cell per colony. I think of this one cell as actually containing the signal, which moves cell by cell back and forth throughout the colony. The signals in all of the colonies are synchronized. The Signal field could consist of  $k$  bits, where  $k$  is some constant that is independent of the colony size. This would allow a count to be kept of how many times the signal has passed through the colony, with the work period ending once the count has reached  $2^k$ . Certain operations within a cell can be triggered when the signal passes through the cell, depending on the count.

In particular, when the signal first passes through a cell, this could cause the cell to copy the Simbit into the Mailbox field. The Simbits would then move one cell at a time, toward the intended neighbor colony. Due to the way in which the initial copying is synchronized, each stream will consist of Simbits alternating with empty spaces. In order to distinguish empty space from real information, the Mailbox field should have two bits: the first bit indicates with a 1 or 0 whether or not the field contains any real information, and the second bit is the actual content; a 00 indicates an empty space. These streams move until the leading Simbit reaches a designated cell in the target colony. Then, one by one, the empty spaces close up and the Simbits in the stream are copied into successive cells. After the first bit of the stream, each subsequent bit can “tell” when it has reached its target destination, because the next cell will have already received its piece of mail.

In a similar fashion, each of the procedures in the simulation can be broken down into a simple succession of copying and comparison operations in a few nested program loops, so that the transition function of  $M_0$  can be constructed in a way that is independent of the transition function of  $M_1$ . By making use of special bit strings to mark certain special locations within each colony, as mentioned above, the transition function for  $M_0$  can also be made independent of the colony size  $Q$  and work period  $U$ . Therefore, the cellular automaton  $M_0$  will be able to simulate an arbitrary cellular automaton  $M_1$ , as long as  $Q$  is large enough to accommodate the look-up table for  $M_1$  and  $U$  is large enough to allow for the number of time steps that it takes to complete one full step of the simulation. In order for  $M_0$  to have the capacity to simulate itself, we merely need to make  $Q$  and  $U$  sufficiently large to accommodate the transition function of  $M_0$ . The vicious cycle mentioned earlier has been avoided, at least for the time being.

This is not the version of self-simulation used by Gacs, and it is also not the one I have in mind for my simplified model. The problem is that noise causes serious difficulties when look-up tables are used, because once a few errors occur in the ProgramBit fields, the look-up table becomes useless. Furthermore, even if we could find a way to maintain the integrity of the look-up table, we would also have a problem when we tried to simulate the local structure transition rules. Among other things, those rules depend explicitly on the Address field, and it takes  $Q \log_2 Q$  bits to list all of the possible values of the Address. This is larger than the colony size, so a look-up table for such rules cannot possibly fit into a colony.

Fortunately, these difficulties can be overcome. Look-up tables are highly inefficient for defining most procedures. For example, it would be ridiculous to use a look-up table to carry out a majority vote procedure involving the Address or Age fields. It would be much better to use a short sequence of copying and comparison operations. Implementing such things efficiently is one of the reasons why programming languages were invented. In Section 7.3 of his paper, Gacs introduces a programming language designed specifically for defining the procedures needed in the transition rules of his cellular automaton. Although I do not have space to give the details, I will try to give you some idea how this programming language solves our dilemma.

Let's go back to the general problem of simulating a cellular automaton  $M_1$  by another cellular automaton  $M_0$ . The role of  $M_0$  will be as before, but the implementation will be different. In particular,  $M_0$  will need to include the local structure described in 5.2. Using Gacs's programming language, write a program for the transition function of the cellular automaton  $M_1$ . This program is a string of bits, and it will replace the look-up table. Write the program, bit by bit, into the ProgramBit fields of  $M_0$ . Now an "interpreter" is needed. This is a special program that can be written in Gacs's (or any other) programming language. It takes as its input the simulated states of 11 cells, together with the program string for the transition function of  $M_1$ , and it gives as its output the output of the transition function of  $M_1$ . The interpreter program is actually a set of simple operations, such as copying and comparing, organized into various programming loops, similar to the set of operations used before in conjunction with the look-up table. These operations are built into the transition function of  $M_0$  so that they are carried out in each colony once during each work period. As before, the transition function for  $M_0$  also implements the various procedures that are required in order for colonies to communicate their simulated states to each other. And also as before, this is done in a way that is independent of the transition function of  $M_1$ , provided the program for that transition function has no more than  $Q$  bits, so that it can be written into the ProgramBits of a single colony.

Now comes the crucial step. The transition function of  $M_0$  can be written as a bit string  $\sigma$  in Gacs's programming language. If  $\sigma$  contains less than  $Q$  bits, then  $M_0$  will be able to simulate itself. The string  $\sigma$  contains three separate pieces. First, it contains the bit string for the interpreter. That part does not depend on the colony size or the work period. Second, it contains strings that represent the transition rules for moving and storing bits, both within and between colonies. As I explained earlier, this part can be made independent of the colony size and work period, but we will soon see that there is no longer any need to do so, as long as the dependence is "mild" in a certain sense. The third part contains the transition rules for the local structure fields. Clearly, this part does depend on the colony size and work period. As with the second part, this dependence is mild.

In order to explain what I mean by mild dependence, I will give a simple illustrative example. Consider a simple majority vote involving the simulated Address fields of five neighboring colonies. This can be carried out by first copying the five Address values into locations in the Workspaces of an interval of cells, and then doing a series of bitwise comparisons. An efficient program for such a procedure involves certain *constants*. These are bit strings that are used to specify logistical details such as the precise location for writing the Address fields of the neighboring

colonies, and the precise time during the work period when the procedure should be initiated. The bit strings for these constants contain some small multiple of  $\log_2 Q$  or  $\log_2 U$  bits. They make up the bulk of the program string for the procedure. The rest of this string contains a few copying and comparison instructions, organized into some program loops. It should therefore be plausible to you that the length of the program string for the majority vote operation is bounded above by  $a \log_2(QU)$  bits, where  $a$  is a constant that does not depend on  $Q$  or  $U$ . This is what I mean by mild dependence.

All of the transition rules described in 5.2 have this kind of dependence, as will the further rules described in 5.4 and 5.5. And even for his more complicated model, Gacs proves in Sections 7.3 and 7.4 of his paper that the string  $\sigma$  is bounded above in length by  $a + b \log_2(QU)$ , where  $a$  and  $b$  are finite constants that do not depend on  $Q$  and  $U$ . In my simple version of the model, I have set  $U = 128Q$ . As I will explain in 5.4, this choice allows enough time during a work period so that the program represented by  $\sigma$  can actually be implemented, at least when  $Q$  is sufficiently large. Thus, if  $Q \gg \log_2 Q$ , there is enough space in a colony to contain a bit-by-bit copy of  $\sigma$ , and enough time during the work period for a simulation step to be completed. In short, our enhanced version of  $M_0$  can simulate itself if  $Q$  is sufficiently large.

I have described how to eliminate the vicious cycle involving the colony size. But there is still one remaining difficulty. How do we maintain the integrity of the ProgramBit field? This is done by “hard-wiring” the program, as I will now explain. For simplicity, assume that the bits in the state of a cell are arranged so that ProgramBit is the first bit and the Address field is the next  $\log_2 Q$  bits. Modify the transition function of  $M_0$  so that near the beginning of each colony work period, the following procedure is carried out within each colony: (i) determine the simulated Address of the colony, as given by the SimBits in the cells with Addresses 1 through  $\log_2 Q$ ; (ii) determine the value of the ProgramBit in the cell whose Address matches the quantity found in the first step; and (iii) write that ProgramBit value into the SimBit of the cell whose Address is 0. Call this modified cellular automaton  $M'_0$ , and let  $\sigma'$  be the bit string that represents the transition function of  $M'_0$ . Determine the initial configuration for  $M_0$  that causes  $M_0$  to simulate  $M'_0$ . In particular, let the initial values of the ProgramBits be given by  $\sigma'$ . This can be done if  $Q$  is sufficiently large, because  $\sigma'$  is not significantly larger than  $\sigma$ . Now let  $M'_0$  start with this same initial configuration. Note that because of the way in which the first SimBit of each colony is overwritten at the beginning of each colony work period, this choice makes  $M'_0$  automatically simulate a copy of itself, with this same initial configuration. Project the resulting time evolution of  $M'_0$  onto the configuration space that is obtained by removing the ProgramBit from each cell, and let  $M$  be the resulting dynamical system. Generally speaking, censoring part of the state space of a cellular automaton in this manner does not give you another cellular automaton, at least not with the same interaction range. But in this case, the ProgramBit of each cell is a function of the Address of that cell. It follows that  $M$  is a cellular automaton with the same interaction neighborhood as  $M'_0$ . Furthermore,  $M$  simulates the specially chosen time evolution of  $M'_0$ , so it simulates itself.

Unlike  $M_0$ , the cellular automaton  $M$  is not capable of simulating a wide variety of cellular automata. But it agrees with  $M_0$  insofar as the transitions of the local

structure fields are concerned, and it simulates itself, without using the ProgramBit field. It is able to carry out this self-simulation by relying on the integrity of the local structure fields.

**Proposition 3.** *Transition rules can be defined for the simulation structure fields so that the deterministic cellular automaton simulates itself, using the colonies as level-1 cells, and using the work period as the level-1 time unit. Included in the transition function of this cellular automaton are all of the local structure transition rules defined earlier. The colonies involved in the simulation are defined by the Address fields, and the work period times for the colonies are defined by the Age fields.*

**5.4. The dynamics of the simulation structure.** Most of the transition rules for simulation structure fields are implicitly defined by the description of self-simulation. The SimBit fields contain the bits of the simulated state, and copies of these bits are passed around by way of the Mailbox fields, so that colonies can interact with other colonies within the level-1 interaction neighborhood. The Workspace fields store these bits once they arrive at their intended destinations. So I will be far less explicit with the transition rules for the simulation structure fields than I was for the local structure fields. Mainly, I will concentrate on details that are needed to make the simulation structure resistant to level-1 noise.

However, before I concern myself with error-correction, there are two special rules that I want to introduce. The first special rule concerns the Mailbox field at a site  $x$ . It demands that all of the Mailbox bits at  $x$  change to 0 if the computed value of Flag1 at  $x$  equals 1. This rule prevents information from passing through blocks of cells where extensive error-correction is going on. It will play an important role in 5.5. The second special rule has a similar purpose. It involves all of the simulation structure fields. It demands that all of the simulation structure bits at  $x$  change to 0 if the computed value of Flag1 equals 1 at  $x$  and the computed value of Address at  $x$  differs from the current value of Address at  $x$ .

For the rest of this section, I will explain how the effects of level-0 and level-1 noise are corrected in the simulation structure fields. I will also define the initial values of the SimBit fields. (The initial values of the Mailbox and Workspace fields were defined earlier to be 0.) To take care of level-0 noise, I use a five-fold redundancy. For example, recall that the SimBit field contains four extra bits, in addition to the primary SimBit. These secondary bits are backup copies of the primary SimBits of the cells at  $x - 2, x - 1, x + 1, x + 2$ . In turn, those four sites contain backup copies of the primary SimBit at  $x$ . At each time step, the five copies of a given bit are replaced by their majority value. In this way, a level-0 error has no real effect on the values stored in the SimBit fields.

I now describe the initial values of the SimBits. As described in 5.3, the primary SimBits in a colony are supposed to represent the individual bits of the state of a single level-1 cell, so the colony size  $Q$  needs to be larger than  $K$ , which is the number of bits in a state. I will assume, in fact, that  $Q \geq 2K$ , so that not all of the SimBits are needed to represent the state of a level-1 cell. Those SimBits that are needed for such a representation will be placed in the colony so as not to be near the colony boundaries. Their initial values are chosen so that the initial state of the level-1 system is the same as the initial state of the level-0 system, with the positioning of the level-1 origin being chosen arbitrarily. In this fashion, all of the initial values of the primary SimBits are determined, except for those that happen

to represent the simulated level-1 SimBits. For these, a simple recursion is required. The secondary SimBits are determined by the condition that they agree with the corresponding primary SimBits.

The Mailbox and Workspace fields involve the same kind of redundancy that was used in the SimBit field. For every bit stored in the Workspace field at  $x$ , there are backup copies in the Workspace fields at  $x - 2, x - 1, x + 1, x + 2$ . And whenever a bit is passed through the Mailbox fields, there are four backup bits moving in parallel at neighboring sites. Majority voting takes place at every time step. And finally, when other types of procedures are carried out, in conjunction with the self-simulation, those procedures are also carried out with the same kind of redundancy. All of this can be organized so that the level-0 noise  $E_0$  has no effect on the simulation structure fields at points  $(x, t) \notin E_0$ .

In addition to the spatial redundancy described in the preceding paragraphs, I also need some temporal redundancy during the colony work period. I divide this work period into 5 stages. The first three stages are comprised of  $U/4$  time units each, and the remaining two stages contain  $U/8$  time units each. Each stage is further divided equally into two parts, an “active period”, which constitutes the first half of the stage, and a “rest period”, which constitutes the remaining half. In the absence of noise, no transitions take place in the simulation structure fields during rest periods.

At the beginning of the active part of the first stage, all of the bits in the Mailbox and Workspace fields are set equal to 0. This procedure is also repeated at the beginning of the active parts of the other stages, except in those parts of the Workspace field that store necessary information gathered during earlier stages. The first three stages are the “information-gathering” stages. During each of them, the colony gets the simulated states from the five nearest colonies on either side, by way of the Mailbox fields. I have allowed enough time for this to take place, since the active period of each of these stages contains  $U/8 = 16Q$  time units. Information gathered during separate stages is stored in separate locations of the Workspace fields. The fourth stage is the “trickle-down” stage. It is only during the first  $2Q$  time steps of this stage that the `Workspace.Flag1` and `Workspace.Flag2` bits are allowed to equal 1. Further detail about the trickle-down stage will be given in 5.5. The fifth stage is the “update” stage. During it, three things happen: (i) bitwise majority votes are taken among the three sets of information that were gathered during the first three stages, (ii) based on the outcomes of these majority votes, the simulation of the transition function takes place, and (iii) the resulting updated simulated state of the colony is distributed bit by bit into the Workspace fields of the cells of the colony. At the beginning of the next work period, all of the SimBit fields are updated, so that they contain this new state. Gacs proves in Sections 7.3 and 7.4 of his paper that no more than  $3Q + a \log_2 Q + b$  time units are required for the procedures of this last stage, where  $a, b$  are constants that do not depend on  $Q$  or  $U$ . Since the active period of the fifth stage contains  $U/16 = 8Q$  time units, there is enough time for the procedures of the fifth stage to be completed if  $Q$  is sufficiently large.

There is one more special procedure that occurs near the end of the active part of the third stage. This procedure can affect the primary SimBits at the cells whose computed Address values are 3 and  $Q - 3$ . It is important for initiating trickle-down, as explained further in 5.5. On a first reading, you may skip this

paragraph. At the end of the third stage, all of the information has been gathered that is needed for computing the level-1 transition. In particular, there is enough information to determine the computed values of the level-1 bits Flag1 and Flag2 in the colony. I will call these computed level-1 bits  $F1$  and  $F2$  respectively. Before the active part of the third stage is completed, I want to change the primary SimBit to  $F1$  at the cell whose computed Address is  $Q - 3$ , and I want to change the primary SimBit to  $F2$  at the cell whose computed Address is 3. For redundancy, the corresponding secondary SimBits are also changed according to these rules. In this fashion, information about the level-1 Flags bits is available to the cells near the ends of the colony before the start of the trickle-down stage. Note that this procedure does not interfere with the simulation, because I have arranged things so that these particular SimBits are not needed in the representation of simulated states.

I will now show how level-1 errors are corrected in the simulation structure fields. I will assume that transition rules have been defined for the simulation structure fields so that the colony work period operates as described in the preceding paragraphs, even in the presence of level-0 noise. Recall that a level-1 error fits into a  $200 \times 200$  space-time box, and that its effects on the local structure fields are limited to two colonies, during a time period of at most  $U/32$ . This time period can touch at most one of the active stages of a colony work period. Outside of the two damaged colonies, the computed values of the local structure and Flags fields are unaffected. And the simulation structure fields of the undamaged colonies can only be affected by possibly incorrect information that flows in through the Mailbox fields. In fact, since at most one of the three information-gathering stages can be disrupted by a level-1 error, the majority vote taken during the fifth stage ensures that the level-1 transitions in undamaged colonies will be based on correct information about the states of other undamaged colonies. Possibly incorrect information about the states of the damaged colonies is treated at level-1 in the same way that a level-0 error is handled in the original level-0 system. It follows that correct level-1 transitions will be made in the undamaged colonies.

What about the simulation structure fields of the damaged colonies? Because of the way in which the Mailbox and Workspace fields are wiped clean at the beginning of each stage, it is not hard to see that the only lasting effects of the level-1 error are in the SimBit fields. At the beginning of the colony work period immediately following the onset of the level-1 error, the SimBit fields of the damaged colonies might represent incorrect states. But in the level-1 simulation, this is treated just like a level-0 error in the original system. It will be corrected in these colonies at the end of that same work period. Thus we have the following result:

**Proposition 4.** *Suppose the errors are restricted to level-1 noise. Then for each level-0 error, there exists a space-time box of the form  $[x, x + 1] \times [t, t]$ , and for each level-1 error, there exists a space-time box of the form*

$$[jQ, (j + 2)Q] \times (kU, (k + 2)U) ,$$

*where  $j$  and  $k$  are integers, such that for all space-time points outside of the union of these boxes, the simulation structure fields have the same value as they would in the deterministic system.*

Note that the space-time box in the proposition contains exactly one time value that is an integral multiple of  $U$ . Thus, the simulated level-1 system corrects level-1 errors at least as well as the level-0 system corrects level-0 errors. Since level-1 errors are much rarer on the time and space scale of the level-1 system than level-0 errors are in the original system (when the noise rate  $\varepsilon$  is sufficiently small), we can conclude that the level-1 system is more reliable in the presence of level-1 noise than the original system is in the presence of level-0 noise.

**5.5. Trickle-down of the Flags field.** Up to this point, I have described a system that is resistant to the damage done by level-1 noise. In the presence of such noise, this system behaves very much like the corresponding deterministic system. Once I define the transition rules for the two bits `Workspace.Flag1` and `Workspace.Flag2`, I claim that the system will also be resistant to level- $k$  noise for all  $k$ . This result is stated in Proposition 5.

Here is some terminology that will be useful in describing some of the problems that can be created by higher level errors. A maximal interval of cells that all have the same apparent colony and the same computed Age value is called a *level-1 cell* if it contains  $Q$  cells. Otherwise, such an interval contains less than  $Q$  cells and is called a *partial level-1 cell*. A full or partial level-1 cell is *aligned* at time  $t$  if its left endpoint is at a site  $x$  with  $x = 0 \pmod{Q}$  and if the common computed value of Age in the level-1 cell equals  $t \pmod{U}$ . It is *misaligned* otherwise. Two full or partial level-1 cells are *relatively aligned* if the locations of their left endpoints differ by an integer multiple of  $Q$  and if they have the same computed Age values. They are *relatively misaligned* otherwise. Similar terminology applies to higher level cells. For example, a level-2 cell consists of  $Q$  relatively aligned level-1 cells, each having the same level-1 apparent colony.

Level-1 errors cannot create misaligned level-1 cells. But a level-2 or higher error can destroy the colony organization in a large region, and then place one or more misaligned level-1 cells into the resulting gap. Such disruptions will not always be corrected by the transition rules of the local structure fields at level-0. Trickle-down is a mechanism that allows the level-1 and higher simulated systems to get involved.

The proof of Proposition 5 is an induction, involving the various levels of the simulation hierarchy. This is one of the places where things work out better with Gacs's version of the model. In his version, relatively misaligned level-1 cells are able to communicate with one another. This allows the simulation hierarchy to continue functioning, even when higher level errors create intervals of misalignment. In my version, relatively misaligned level-1 cells are prevented from communicating with each other, because the inevitable presence of an inconsistency between them produces an interval of 1's in the `Flag1` bit. (Recall from 5.4 that 1's in the `Flag1` bit cause the `Mailbox` field to get erased.) This is actually the way it was done in Gacs's 1986 version. The reason I did not follow Gacs's current approach is that it would have made the transition rules more complicated. Either way, I would have had to leave out some of the details. Nevertheless, I will explain all of the key ideas, and I will make it clear where I am leaving things out. There is a summary at the end that you might find helpful when the going gets tough.

I wish to show inductively that the damage done by a level- $k$  error  $S$  to the level-0 local structure fields is limited to a space-time box of size  $2Q^k \times U^k/32$ . The case  $k = 1$  was done in 5.2. By reasoning as I did there, I can assume that the noise

is restricted to level- $(k - 1)$  outside of  $S$ , since all other level- $k$  or higher errors are too far away to have an effect.

To simplify the discussion, I will restrict my attention to  $k = 3$ . For this case, the inductive hypothesis is the following: if the noise is restricted to level-2, then all damage to the level-0 local structure fields is limited to a collection of space-time boxes, where the sizes of the boxes depend on the levels of the errors that are associated with them. In particular, the level-2 errors are associated with boxes of size  $2Q^2 \times U^2/32$ .

To begin the argument, I define the damage island associated with  $S$ . At any given time, it is the smallest interval that contains all cells where the level-0 local structure fields have computed values that are different from what they would be if  $S$  were removed. My main goal is to explain how this damage island disappears, just as I did in the case  $k = 1$ . Roughly speaking, the reasoning will be the same. I will argue that the left end of the island moves to the right at some minimal positive average speed, and that the right end of the island stops moving near the left end of an undamaged level-3 cell.

There are two main reasons why this argument doesn't work as it stands. The first is that the damage island can be quite large, several hundred times  $Q^2$ . Part of the argument in 5.2 relied on the island being smaller than size  $Q$ . This problem affects the erosion at the left end of the island. The other problem involves the right end. It is true that the right end gets "stuck" at the left boundary of a level-3 cell, since the Flag1 bits will normally be 0 within the level-3 cell, even if the damage island lies immediately to its left. But a few well-placed level-1 or level-2 errors can get the right end "unstuck". There are a few other minor difficulties, such as occasional growth at the left end of the island due to level-2 noise, but these are not very serious. Once you understand how the two main problems are solved, you understand the heart of the proof.

The problem of the left end is the easier of the two. Typically, this end moves steadily to the right until it reaches the left end of a level-1 cell. Since the Flag1 bits can equal 0 in the underlying level-0 cells, the erosion might stop. Trickle-down gets it going again, as I will now explain.

For the moment, let me assume that the level-1 cell at the left end is not part of a level-2 cell in the island. Because of the inconsistency at the left end, there must be an interval of 1's in the Flag1 bits of several cells immediately to the left of the island. These 1's prevent any information from passing between the inside and outside of the island, due to the special transition rules for Mailbox introduced in 5.4. This situation forces a level-1 inconsistency in the level-1 cell at the left end, causing the level-1 Flag1 bit in that cell to change to 1. As I will explain in more detail when I give the transition rules for Workspace.Flag1, this change eventually causes 1's to flow into the level-0 Flag1 bits of all of the underlying level-0 cells. (This kind transfer of a Flags bit from one level of the simulation hierarchy to a lower level is what I am calling "trickle-down".) When these level-0 1's reach the left end of the island, the erosion starts up again. This procedure occurs once every  $U$  time steps, during the trickle-down stage of the work periods of the level-1 cells, and it lasts long enough to move the left end of the island at least one full level-1 cell to the right. There are occasional reversals, due to level-1 errors, which can occur once every 24 work periods, and due to level-2 errors, which can occur once every  $24U$  work periods. Trickle-down of the Flag2 bit can also cause delays, but only

once every 16 work periods. I claim that such delays can be considered “transient effects”, similar to the ones that were discussed in 5.2. They are not enough to significantly slow the steady erosion at the left end. (This is one of the places where I am going to skip the details.) In spite of the delays, the erosion advances at an average speed that is at least  $Q/2$  level-0 cells in every  $U$  time steps, or  $1/2$  level-1 cell per level-1 time unit.

Now what happens if the left end of the island coincides with the left end of a level-2 cell? Such a high degree of organization within the island can prevent the level-1 Flag1 bits from changing to 1’s. But unless this level-2 cell is part of a full level-3 cell, its level-2 Flag1 bit will eventually change to 1. Because of the simulation hierarchy, which operates at least at level-1 inside the level-2 cell, this value trickles down to the underlying level-1 cells, after which it trickles down to level-0. As a result, the erosion of the left end continues. (There are some simple details here that need to be checked concerning the coordination of the trickle-down at various levels.) After taking into account the transient effects of the Flag2 bit at various levels and also the level-2 noise, it can be shown that this multi-level trickle-down is sufficient to cause the left end of the island to move at least  $Q^2/2$  cells to the right every  $U^2$  time steps, on the average. Thus, at the scale of level-2, the left end erodes at an average speed of at least  $1/2$  cell per time step. This behavior is analogous to what happened at level-0 in the proof of Proposition 2. And just as in that proof, I do not need to worry about the possibility that a level-3 cell stops the motion of the left end, since the island will never be able to grow large enough to accomodate such a cell. As far as the left end of the island is concerned, the argument is finished. I have left out some parts, but I ask you to believe that I have not hidden anything important.

Note that the erosion of the left end of the island can be much faster than my lower bound. If the cells in the island are not highly organized, the erosion can be just as fast as it was in the proof of Proposition 5.2. If this is the case, then my problem with the right end of the island is not very serious. The presence of undamaged level-1 cells outside of the island slows the movement of the right end enough so that I can be sure that the left end catches up with it. For this reason, I can assume in my discussion below that the cells in the colony are fairly well organized. (In a complete proof, I would need to be more precise about this point.)

The problem of the right end of the island is a little bit tricky. As I said earlier, once the right end reaches an undamaged level-3 cell, it is possible for level-1 and level-2 errors to cause the island to expand further to the right. I will use trickle-down of the Flag2 bit to keep the island from moving more than two level-2 cells beyond the left boundary of the level-3 cell. Once I accomplish this, the argument is finished.

Here is the picture I have in mind. Suppose the right end of the island is positioned precisely at the left boundary of an undamaged level-3 cell. Further suppose that there are several relatively aligned level-2 cells at the right end of the island. The rightmost one of these cells may be partial, and for simplicity, I will assume that it is partial. Finally, suppose that a level-2 error occurs in the level-2 cell that lies immediately to the right of the island. If 1’s in the level-1 Flag1 bit spread throughout this level-2 cell, the partial level-2 cell in the island may be able to grow into a full level-2 cell. Depending on the positioning of the error, further growth of the island is possible, as much as two full level-2 cells. After  $24U^2$  time

steps, another level-2 error could cause a similar expansion of one or two level-2 cells. Before long, the island could expand far enough so the level-3 cell no longer acts as a deterrent to further expansion. Before this can happen, I need to reverse the motion of the right end of the island and restore the left end of the level-3 cell. This is the reason for Flag2.

Consider the rightmost three full level-2 cells in the island. The special transition rule for Mailbox prevents any information from passing through the partial level-2 cell at the extreme right end of the island, so these three full level-2 cells receive no communication from outside of the island. As a result, the level-2 apparent colony of these cells does not exist. If the level-2 Age in these cells is divisible by 16, condition (iii) in the transition rule for Flag2 is satisfied at level-2, causing the level-2 Flag2 bit in these cells to become 1. This value trickles down to level-0. (I will give more detail about this when I define the transition rules for Workspace.Flag2.) At the same time, there is trickle-down in the Flag1 bit, due to the level-2 inconsistency at the right end of the island. Because of the way in which the Flag2 bit reverses the direction of error-correction, the right end of the island moves towards the left. I have arranged things so that it moves left two full level-2 cells during the trickle-down stage of the work period of the level-2 cells. Since this procedure occurs every  $16U^2$  time steps, and since level-2 errors can only occur once every  $24U^2$  time steps, the direction-reversal is just enough to undo any expansion of the island that might be caused by level-2 errors. Similar things happen at level-1, to reverse damage done by level-1 errors. In this way, the island is not allowed to expand very far beyond the left end of the level-3 cell. Therefore, the left end of the island catches the right end, and the island eventually erodes away completely. As order is restored to the local structure at level-0, the simulation hierarchy makes the same thing happen at higher levels. I claim that the island is eliminated within  $U^3/32$  time units, and that the level-1 and level-2 organization is restored within another  $U^3/32$  time units. Residual effects in the simulation structure and Flags fields are eliminated in a manner that is similar to what happened in the proof of Proposition 2.

Note that in this argument, we do not need to worry that a 1 in the Flag1 bit might trickle down from level-3 or higher inside the level-3 cell. Since the damage island is eliminated quickly on the time scale of level-3, the damage from  $S$  is treated like a level-0 error in the level-3 simulation. Thus, it cannot cause the level-3 Flag1 bit to change to 1 in the level-3 cell. And this damage is “invisible” at higher levels.

I have left out a lot of details in this argument. Some of the strange features in my transition rules are necessary because of these details. But they should not detract from the main ideas, which are explained above. There are many different ways to implement those main ideas. Gacs’s model is one of those ways, my version is another.

Before I state Proposition 5.5, I will provide further details about the trickle-down procedures, including the transition rules for Workspace.Flag1 and Workspace.Flag2. You may want to skip these details on a first reading. Here are the transition rules for Workspace.Flag1:

$0 \rightarrow 1$  if all three of the following conditions hold: (i) the computed value of Address at  $x$  is in the interval  $[Q - 5, Q - 1]$ , and (ii) the computed value of Age at  $x$  is in the interval

$[3U/4, 3U/4 + 2Q)$ , and (iii) the computed value of SimBit at the site  $(x - A(x)) + (Q - 3)$  is 1, where  $A(x)$  is computed value of Address at  $x$

$1 \rightarrow 0$  if at least one of conditions (i), (ii), or (iii) fails.

The first two conditions say that this bit can normally be 1 only in the rightmost 5 cells of a colony, and only during the first  $2Q$  time steps of the trickle-down stage. Assuming the computed values of Address indicate the correct position within the colony, the site referred to in condition (iii) has computed Address  $Q - 3$ . According to the special transition rules for SimBit in 5.4, condition (iii) typically holds after the third stage of the colony work period if the level-1 computed value of Flag1 is going to be 1 at the end of the work period. When all three conditions hold, the Workspace.Flag1 bits in the five cells at the right end of a colony change to 1's, and immediately thereafter, the corresponding Flag1 bits change to 1's, thereby initiating a leftward-moving wave of 1's that moves throughout the colony. This trickle-down process gets "turned off" after  $2Q$  time steps. If the colony is still intact after that time, the Flag1 bits change back to 0's in a leftward-moving wave that starts at the right end.

Here are the transition rules for Workspace.Flag2.

$0 \rightarrow 1$  if all four of the following conditions hold: (i) the computed value of Address at  $x$  is in the interval  $[0, 4]$ , and (ii) the computed value of Age at  $x$  is in the interval  $[3U/4, 3U/4 + 2Q)$ , and (iii) the computed value of SimBit at the site  $x - A(x) + 3$  is 1, where  $A(x)$  is the computed value of Address at  $x$ , and (iv) the computed value of Flag1 at  $x$  is 0

$1 \rightarrow 0$  if at least one of conditions (i), (ii), (iii), or (iv) fails.

Note the similarities and differences between these rules and the ones for Workspace.Flag1. The Workspace.Flag2 bits are only allowed to be 1 near the left end of the colony, rather than at the right end. Condition (ii) and (iii) are the same, except that the site involved in (iii) has computed Address value equal to 3. According to the special transition rules for SimBit in 5.4, this condition for Flag2 typically holds after the third stage of a colony work period if the level-1 computed value of Flag2 is going to be 1 at the end of the work period. Condition (iv) has no analogue in the transition rules for Workspace.Flag1. The details of these rules have been chosen so that the trickle-down can be properly coordinated between various colonies. When all four conditions hold, a process is set in motion that is similar to what happens with Flag1, except that the Flag2 process starts at the left end of the colony. Both Flag1 and Flag2 trickle-down can occur simultaneously, as shown by the discussion below.

To help you understand these rules, let me describe a typical example of trickle-down. Consider a level-1 cell whose computed level-1 Flag1 and Flag2 bits are both equal to 1. According to special procedures that were introduced in 5.4, this information is stored near the ends of the level-1 cell, during the third stage of the work period. At the beginning of the trickle-down stage, the Workspace.Flag2 bits change to 1 in the five level-0 cells at the left end of the level-1 cell, and the Workspace.Flag1 bits change to 1 in the five level-0 cells at the right end of the level-1 cell, as specified in the transition rules for Workspace.Flag1 and Workspace.Flag2.

Immediately thereafter, a block of five 1's appears in the Flag2 bits at the left end of the level-1 cell, and a block of five 1's appears in the Flag1 bits at the right end. These blocks initiate waves that travel into the level-1 cell from opposite directions. The Flag1 wave is a little faster, so it covers the level-1 cell before the Flag2 wave does. When the Flag1 wave reaches the left end of the level-1 cell, the Workspace.Flag2 bits there change back to 0, because of condition (iv) above. The block of 1's in the Flag2 bit begins to slowly erode at the left end. This block grows faster at the right end than it erodes at the left, so it continues to grow in length for a while, at least until it reaches the right end of the level-1 cell.

Now suppose that this type of trickle-down occurs simultaneously in three consecutive level-1 cells. The presence of 1's in the Flag1 bit allows the blocks of 1's in the Flag2 bit to cross from one level-1 cell to the next. It is not hard to see that the Flag2 blocks therefore merge at some point, covering at least two of the three level-cells. This single large block continues to grow in size as it drifts to the right. If these three level-1 cells are at the right end of a damage island (as was the case in the proof of Proposition 5), then the Flag2 block will eventually move beyond the right end of the island. After that, majority voting from the right will cause the right end of the island to drift to the left, as described earlier. The timing is such that two new level-1 cells are created. These new cells are aligned with the level-1 cells outside of the island. Initially, the simulation structure fields in these new cells are blank, ensuring that the Flags fields return to value 0 after a short time period. Assuming that these cells lie at the left end of a level-2 cell (as they do in the proof), the blank SimBit fields will be corrected after one work period, by way of the level-1 simulation.

This example only illustrates trickle-down from level-1. The proof of Proposition 5 requires trickle-down from all levels. The handling of newly created colonies gets a little tricky when trickle-down occurs at multiple levels, especially when it involves the Flag2 bit. But I have worked out the details for myself. You will have to trust me here.

**Proposition 5.** *For  $k \geq 0$ , each level- $k$  error can be enclosed in a space-time box of the form  $[jQ^k, (j+2)Q^k) \times [t, t+U^k/16)$ , where  $j$  is an integer, such that outside of the union over  $k$  of these boxes, the level- $n$  Address fields agree with their initial values for all  $n \geq 0$ .*

The rest is easy. Proposition 1 implies that, for small  $\varepsilon > 0$ , most space-time points are not in any of the boxes described in Proposition 5. So the initial values of the Address fields are preserved with high probability for all time, as desired.

**Summary.** The above argument is supposed to show how the system is able to distinguish a finite island (created by errors) from the surrounding "sea". The key point is that all interfaces between the sea and an island tend to drift right, at an average rate that depends on the level of colony organization on the right side of the interface. Because an island is finite in size, its internal colony organization breaks down at some level. Trickle-down of the Flag1 bit communicates any high-level lack of organization to level-0. Within the sea, the colonies are organized to an arbitrarily high degree, so trickle-down does not take place as often. As a result, the "sea-island" interface drifts to the right faster than the "island-sea" interface. The difference in speeds is very slight if the island is large, so some islands survive a very long time. But islands of similar size are far enough apart so that they are eliminated before they can interact with each other. This is the reason

that the colony organization in the sea is preserved at all levels. When a large island interacts with a smaller one on its left, the effect is not significant. When a large island interacts with a smaller one on its right, the effect is counteracted by trickle-down of the Flag2 bit.

It is interesting to note that the size of the noise rate  $\varepsilon$  plays very little role in the argument. It only enters into the proof of Proposition 1. A similar statement can be made about Gacs's version. I mention this fact because it helps illustrate the high degree of robustness that is generally present in Gacs's counterexample to the Positive Rates Conjecture.

**Acknowledgment.** I wish to thank Eugene Speer for numerous valuable comments and suggestions. He found serious mathematical errors in early drafts of this paper, and the article is vastly improved because of his careful work.

#### REFERENCES

- [1] J. van den Berg and H. Kesten, *Inequalities with applications to percolation and reliability*, Journal of Applied Probability **22** (1985), 556–569.
- [2] P. Gacs, *Reliable computation with cellular automata*, Journal of Computer System Science **32** (1986), 15–78.
- [3] P. Gacs, G. Kurdyumov, and L. Levin, *One-dimensional homogeneous media dissolving finite islands*, Problems of Information Transmission **14** (1978), 92–96.
- [4] L. Gray, *The positive rates problem for attractive nearest-neighbor systems on  $\mathbb{Z}$* , Z. Wahrscheinlichkeitstheorie verw. Gebiete **61** (1982), 389–404.
- [5] L. Gray, *The behavior of processes with statistical mechanical properties*, Percolation Theory and Ergodic Theory of Infinite Particle Systems, Springer-Verlag, New York, 1987, 131–167.
- [6] G. Kurdyumov, *An example of a nonergodic homogeneous one-dimensional random medium with positive transition probabilities*, Soviet Mathematical Doklady **19** (1978), 211–214.
- [7] T. Liggett, *Interacting Particle Systems*, Springer-Verlag, New York, 1985.
- [8] K. Park, *Ergodicity and mixing rate of one-dimensional cellular automata*, Ph.D. Thesis, Boston University, Boston, MA 02215, 1996.
- [9] A. Toom, *Stable and attractive trajectories in multicomponent systems*, Advances in Probability, vol. 6, Dekker, New York, 1980, 549–575.

SCHOOL OF MATHEMATICS, UNIVERSITY OF MINNESOTA, MINNEAPOLIS, MN 55455  
*E-mail address:* `gray@math.umn.edu`