Gács' Cellular Automata

東京大学大学院総合文化研究科 D2
行田悦資*

1 はじめに

ノイズあるいはエラーは、まともな (正しい) 計算をしようとする場合には非常に厄介な存在です。現実の計算機の設計においては常にノイズの影響を隠蔽するしかけを組み込ませる必要があります。Reliable Computing とは、このようなノイズ環境下においても信頼のおける計算を行うにはどうしたらよいのか、ということを考える分野なのです。本稿では、2001 年に発表されて話題を呼んだ Gács の確率セルオートマトンを紹介しようと思います。

まずはざっと背景的なことを含めて全体像を説明していきましょう。

Positive Rate Conjecture と呼ばれる、ある数学上の予想があります。簡単に言うと、1 次元の確率セルオートマトンではノイズ環境下でも初期配置を記憶しつづけられるようなルールは構成できないであろう、という予想です。この場合、1 次元というのがポイントで、2 次元以上のセルオートマトンでは成立しません。1 次元の場合には後述するようにある特殊な事情があるので、その主張はもっともらしいと思われていました。したがって、ほとんどの研究者にとっての主な関心はどうやってその予想を証明するのか、ということにあったようです。ところが 2001 年に Peter Gács が、この予想に対する反例を構成してしまって大きな話題を呼んだのです。話題になった理由のひとつはその論文 [1] の分厚さ (200 ページを越える!) が尋常でないことと、しかもその前座には Gray というレフェリーによるリーダーズガイド [3](これも４０ページある) までついていることです。本稿もそのリーダーズガイドと Gács 自身が別のところで発表している講義録 [2] を参考にして書いています。

Gacs が構成したのは「エラー修正を自己組織的に行うセルオートマトン」で、その基本的なアイディアはコロニーと呼ばれるエラー処理のモジュールを再帰的に構成するところにあります。コロニーとは、ある一部のセルにノイズが入ってエラーが起こったとしても、計算結果を相互に監視、修正しあって、その一部のセルのエラーが全体に波及しないようにさせた、セルのかたまりです。もちろん、ひとつのコロニーだけでは対処できないような広範囲にわたるエラー (バーストエラー) が起こりうる危険もあります。そうした場合には、コロニーを単位としたかたまりであるメタコロニーを構成しておいてあげれば処理ができるでしょう。さらにそれでも駄目ならメタコロニーのかたまりであるメタメタコロニーを構成して... という風に、エラー修正コロニーの連鎖を無限に再帰的に構成していけばよいであろう、というのが Gacs のアイディアなのです。

ところで、こうしたエラー修正モジュールを再帰的に構成するというアイディアは、1950 年代に von Neuman の Boolean Circuit の研究の中に見ることができます [4] [2]。Neuman のモデル

* *E-mail*: nameda@sacral.c.u-tokyo.ac.jp

と Gacs のモデルの大きな違いは、Neuman のモデルが再帰的構造を外から (静的に) 与えるのに対して、Gacs のモデルでは再帰的な構成の連鎖をローカルルールだけを使って自己組織的に (動的に) 構成していくところにあります。では、まず最初に Neuman の Boolean Circuit の説明をして、その後で Gacs のセルオートマトンを見てゆくことにしましょう。

2 Boolean Circuit

もともと von Neuman は生物の情報処理過程に大きな関心を寄せていて、その論文の中で逆接的ですが、「生物はノイズを抑えると言うよりむしろ積極的に利用しているように見える」、と感想をもらしています。しかしながら、ノイズを逆手に取る方法を考えるというのは大変難しいわけで、とりあえずは不安定な素子からいかに安定な出力を得るか、ということを考えようとしたようです。

Boolean Circuit とは、その名の通りいくつかの Bool 演算素子を組み合わせた回路です。素子には入力ノードと出力ノードがあって、それぞれが Boolean 関数として働きます。図 1 に見るように、それらの素子を階層的に組み合わせて全体としての関数の出力を得ます。なおここでは、ある素子の出力がそのまま自分自身への入力となるようなサイクリックな回路は考えません。

こうした回路で計算を行うと、図 1(B) のようにある素子で発生したエラーが時間とともに回路全体へと波及してしまう恐れがあり、結果として得られる出力は全く信頼できないものになってしまいます。これをなんとか、回路全体の出力のエラー確率を (0 にはできないかもしれないけれども) ある値 δ 以下に押えられるようにする方法を見ていきましょう。

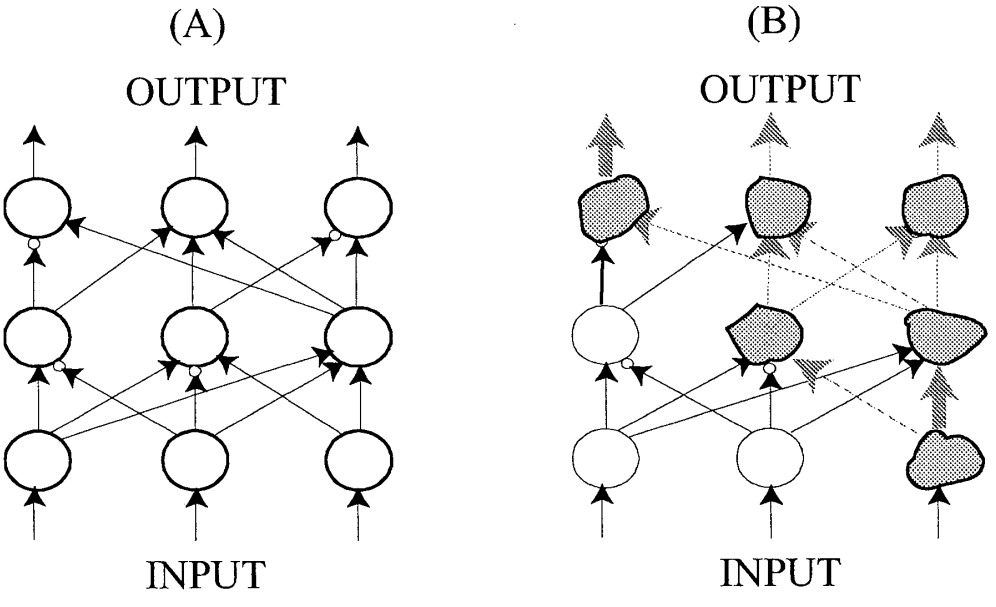
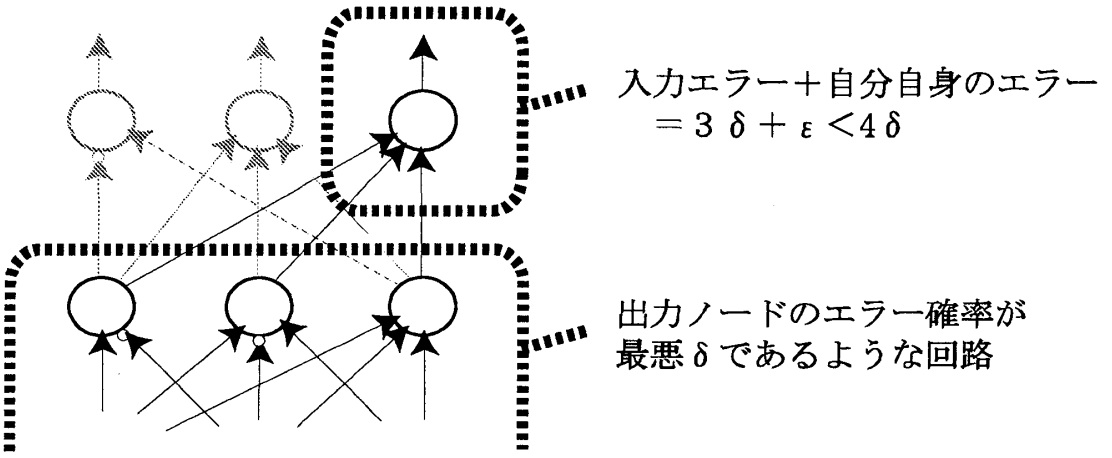


図 1: (A) Boolean Circuit の例。それぞれの素子では $\wedge$ (and) や $\vee$ (or) などの基本 Bool 演算やその組み合わせ演算を行う。出力が複数ある場合には同一の出力結果が次の全ての素子へとつながる。(B) 右下の素子で発生したエラーが時間とともに伝搬してしまう。

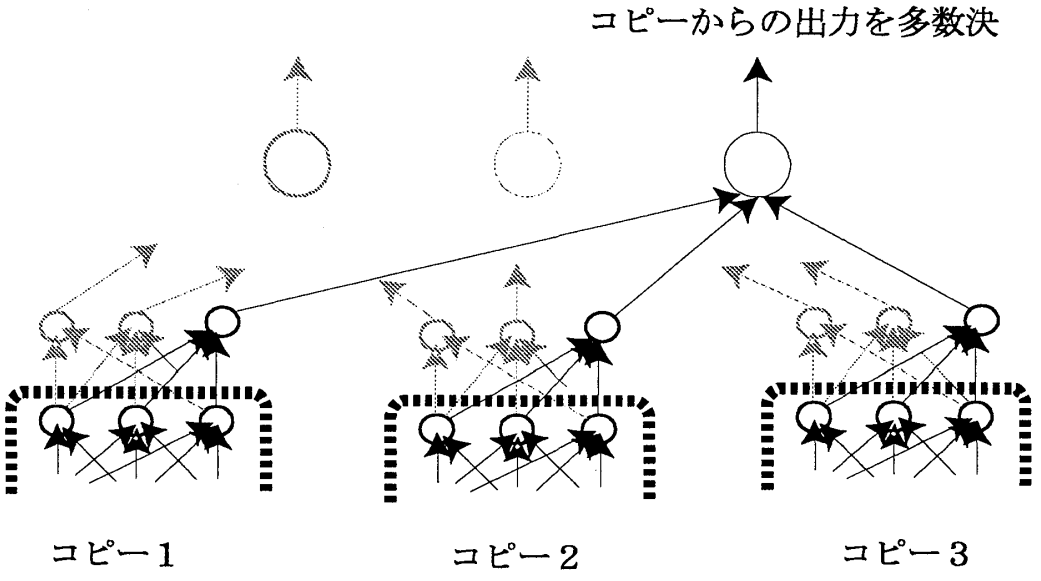
ある素子でエラーが発生する確率を ϵ とします。十分小さな $\epsilon > 0$ に対して、どんな Boolean Circuit もそれと等価で出力のエラー確率が $\delta < \frac{1}{2}$ であるような回路を構成できることが証明できます [2]。本当はエラー確率について、もう少しきちんとした定義を与えなければいけないのですが、繁雑になるのでここでは省略します。また、各素子への入力ノードの数も 3 つまでに制限します。

構成は以下のように再帰的に行います。

いま、ある回路の最上段の素子について考えます。その素子より下にある回路からの入力エラーが δ 以下であったとすると、$\delta < \epsilon$ ならば、自分自身の出力エラー確率は最悪 $3\delta + \epsilon < 4\delta$ になります。



この増えた分の出力エラーを補正するために、まったく同じ回路のコピーを作って 3 つのコピーによる多数決を行います。こうして作られた回路はもとの回路よりも段数が 1 つ増えます。



p この新しく作った回路の出力エラー確率は、最悪

$$3(4\delta)^2 + \epsilon = 48\delta^2 + \epsilon < 49\delta^2$$

となります。したがって、$49\delta^2 < \delta$ かつ $\epsilon < \delta^2$ を満たすような δ を選ぶならば、この回路の出力

エラー確率は最悪 δ で抑えることができます。このようにして図中の点線で囲った部分についても同様の操作を行い、最終的には最下段までたどり着くまで再帰的に回路を構成してゆけばよいわけです。

当然ながらこの構成方法では、もとの回路の 2 倍の段数が必要になります。また必要な素子の数はもとの回路の段数を d とすると、3^d 倍に爆発的に膨れ上がってしまうという重大な欠点があります。もちろん、ここで紹介した方法はもっともシンプルな方法であって他にも賢い方法があります。[2] には素子数の増大を定数倍程度にまで抑える方法が紹介されています。

ともあれこれ、これでノイズによる影響を抑えてある程度の信頼性を得られる回路が構成できたわけで、この構成方法の特徴はエラー修正する多数決回路を組み込んで、再帰的な構成を行う点にあります。この特徴は Gács のモデルでも同じですが、Boolean Circuit では再帰的な構成をハードウェア的に仕込む必要がある、というわけです。

3 確率セルオートマトン

この節からはセルオートマトンにおける Reliable Computation を見ていきます。前節での Boolean Circuit との違いは、Boolean Circuit がなんらかの関数の出力を安定させる目的で考えられているのに対し、セルオートマトンでは初期状態を記憶し続けること (メモリー) に注目をおいています。Gács のセルオートマトンのモデルは 1 次元で動作することが新しいと、冒頭で書きました。なぜ 1 次元では困難だと思われていたのか、いくつかの例を見ながら説明していきます。なお、以下では有限状態数で、相互作用距離が有限の確率セルオートマトンを扱っていきます。特にことわりの無い限り、状態数は $\{0,1\}$ の 2 つで、系の大きさは無限とします。

確率セルオートマトンとは状態遷移規則にしたがって遷移するはずが、確率 ϵ でまったくデタラメな状態に遷移してしまうようなセルオートマトンです。確率セルオートマトンの研究は、伝統的に統計力学や熱力学の範疇にあります。そこでは異なる独立した初期状態から始まって、長い時間が経た後の系の状態が同じ状態に収束するかどうかを問題にします。もし全ての初期状態が同じ状態に落ち着くならば、定常状態がただ一つしかないということになります。別の言い方をすれば、系は初期状態を「忘れて」しまったと言えるでしょう。この場合の「状態」とは状態配置のことではなく、系全体での平均的な指標 (例えば状態 0 にあるセルの数の平均など) を見ています。

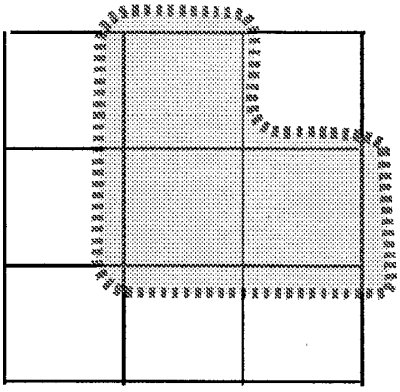


図 2: Toom の 2 次元セルオートマトン。中央が自分のセル。点線で囲った部分のセルの状態を見て多数決を取って次の状態を決める。

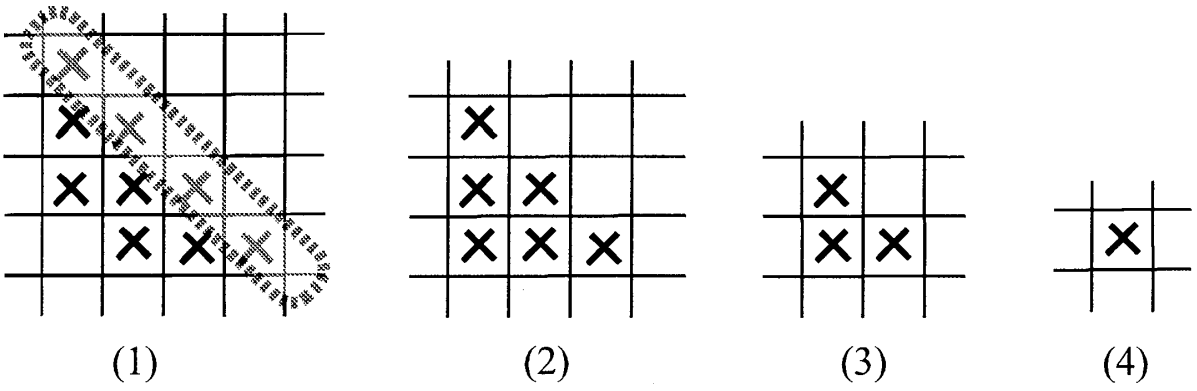


図 3: Toom のセルオートマトンがエラー修正を行う様子。×印がエラーの起こったセル。(1) では点線で囲まれたセルが次のステップでエラー修正される。以下 (2)〜(4) まで左下方向に向かって順次エラーが修正され、最後のステップでは完全にエラーが消滅する。

はじめに Toom の 2 次元セルオートマトンの例を見てみましょう (図 2)。Toom のルールは自分とその上と右の 3 つのセルの状態で、多数決をとって次の状態に遷移します。このセルオートマトンルールはエラーに対して非常に強い性質を持っていて、図 3 にそのエラー修正の様子を示しています。このエラー修正は、初期状態で全てのセルが 1 であるとき (**ALL 1**) も、全てのセルが 0 であるとき (**ALL 0**) でも同じように動作します。すなわち、十分に小さいエラー確率 ϵ のもとでは、Toom のセルオートマトンは **ALL 0** と **ALL 1** の 2 つの初期状態を「記憶」し続けることができて、互いに混じり合うことはありません。

次に統計力学での強磁性 Ising モデルの例を見てみます。強磁性 Ising モデルでは、2 次元ではスピンが全て上向きの状態 (**ALL+1**) と、スピンが全て下向きの状態 (**ALL-1**) が、十分に小さな温度 (=エラー確率) では保存され続けます。これは、図 4(A) に見られるように、エラーの島に比例して「境界」も大きくなり、エラーの境界では相互作用エネルギーを損するため、エラーの島を小さくする作用が働くためです。しかし、図 4(B) のように 1 次元の強磁性 Ising モデルでは、いつまで経っても境界はただの点なので大きくならず、2 次元のときのようなエラーを小さくする作用は働きません。したがって 1 次元では初期状態が何であれ、ただひとつの平衡状態「半々」の状態へと系は向かってしまいます。

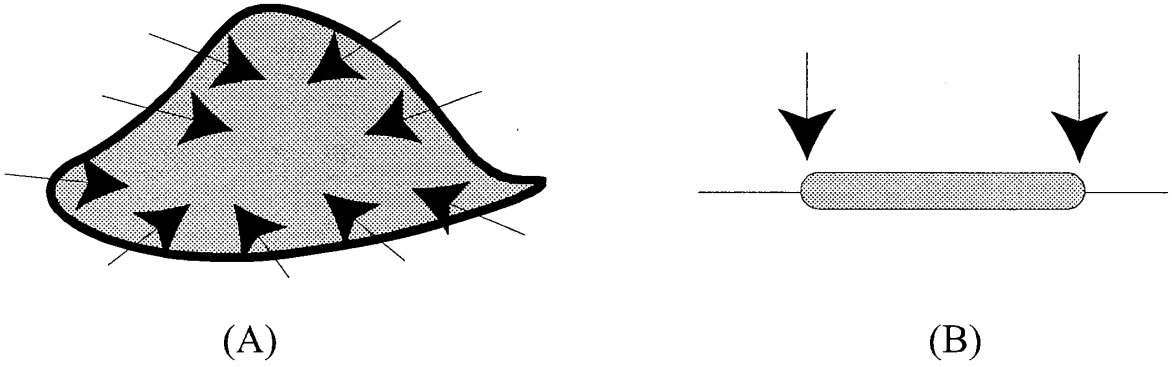


図 4: (A) 2 次元ではエラーの島に比例して境界も大きくなる。(B) 1 次元では境界はただの点にすぎない。

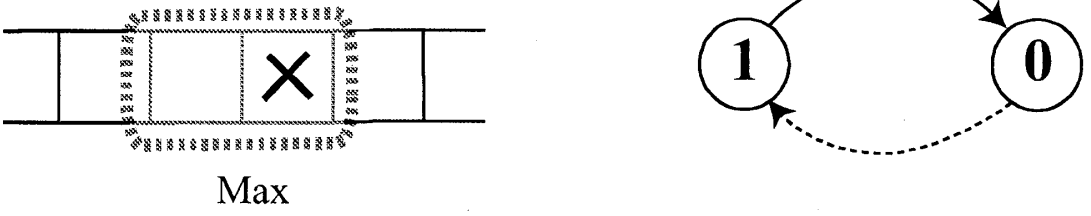


図 5: 1次元方接触モデル。自分と左隣のセルで大きい方の値を採用する。右図の状態遷移図に示したように、状態 1 は確率 ϵ で遷移する可能性があるが、状態 0 は普通の状態遷移規則にしたがって遷移するときだけ状態 1 になる。

もっとも強磁性 Ising モデルには、「単調性の存在」という特殊な事情があります。単調性 (強磁性) とは、どこかのセルを (-1) から $(+1)$ に変えたとすると、将来に $(+1)$ を多く見られる可能性が増える、という性質です。そうした特殊事情を抜きにしても、1次元のセルオートマトンによるエラー修正には「境界」の問題が常に付きまとってしまいます。Toom のルールをいくら改造しても1次元ではうまく動きません。

ところで、状態 $\{0,1\}$ のうち片方の状態ではエラーが発生しないという「ずるい」ルールを採用すると、1次元のセルオートマトンでも2つの平衡状態を持つことができます。これが片接触モデルと呼ばれるルールで、図5に示しているように状態 0 はノイズの影響を受けません。そうすると、**ALL 0** と **ALL 1** の2つの状態は $\epsilon > 0$ でも、互いに保存されたままになります。

その他にも、相互作用距離を動的に変化させるという方法も考えられますが、ここでは総合作用距離を有限に縛っているので考えません。そうした一切の「ずる」をしないで Reliable Computing を実現する1次元のセルオートマトンは構成できないであろうという予想が Positive Rate Conjecture と呼ばれる予想でした。Positive Rate とは、どの状態に対してもある確率 $\epsilon > 0$ でエラーが起こりうる、という意味です。この予想が Gács によって否定的に解決されたのでした。

4 自己シミュレーション

Gács は1次元における境界の問題を、コロニーと呼ばれるセルの集合体の再帰的構成を使って回避しています (図6)。こうしたコロニーの無限の再帰的構成を有限相互作用距離だけを使って行うことを可能にしたのが、自己シミュレーションという手法です。

まずはコロニーの自己シミュレーションのしくみの前に、単位セルがどのように構成されているかを見てみましょう。Gács のルールにおける状態数はかなり大きく、実在の CPU のレジスターのように 2bit で表現され、役割が規定されていています (図7)。実際のところ論文 [1] は、論文というより RISC プロセッサーの仕様書に近いです。初期状態としては **ALL 0** や **ALL 1** のような単純なものではなく、各セル毎に少しずつ異なった状態でスタートします。

セルの状態の2進数表記を大まかに分解すると4つの部分になります。図7における各ビット列の意味は次のとおり。

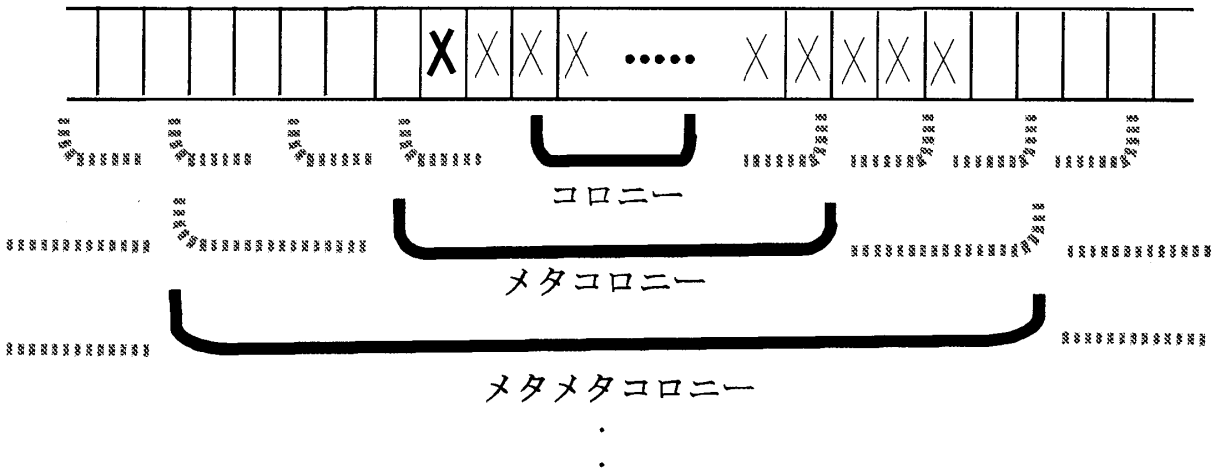


図 6: コロニーはセルの集合体であり、エラー処理はコロニー単位で行われる。しかし、ひとつのコロニー内で処理できないような広範囲にわたるエラーにはコロニーのコロニー (メタコロニー) 単位で処理が行われる。さらにメタコロニーでも処理できなければメタメタコロニーが... 無限に再帰的な連鎖は続く。

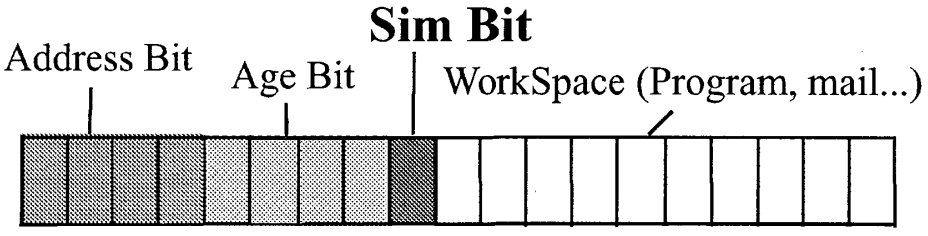


図 7: セルの状態の 2 進数表記。セルの状態数を 2^k とすると、k bit で表現する。

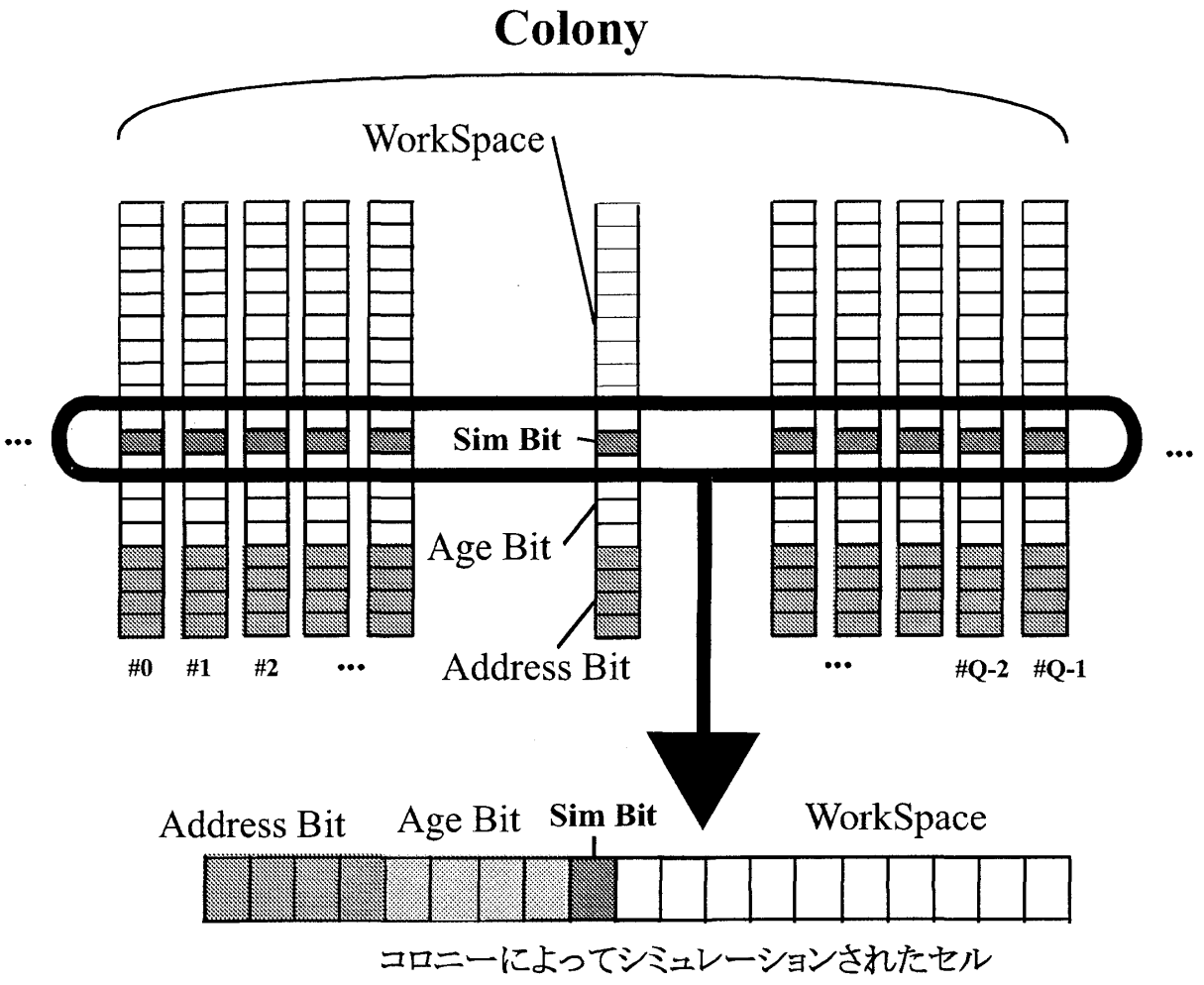


図 8: コロニーによる自己シミュレーション。各セルの状態 bit 列を縦に表記してある。#記号はセルのコロニー内でのアドレス。Sim Bit を繋ぎ合わせて眺めると、あたかもひとつのセルであるかのように振舞う。このシミュレーションされたセルの Sim Bit を再び繋ぎ合わせると、今度はコロニーのコロニーによる自己シミュレーションが実行されている。

Address Bit セルのコロニー内における位置を表現する。コロニーの大きさを Q とすると、0〜$Q-1$ までの数字が繰り返される。

Age Bit 1 ステップ毎に更新されるカウンター。コロニーの活動周期 U を表現する。1 順する毎にコロニーは 1 ステップ更新する。

Sim Bit 自己シミュレーションのためのビット。

WorkSpace 警報 Flag や、自己シミュレーションのためのプログラム、離れたセル同士が交信するためのメールを保存するビットなど。

です。それぞれの Bit は、各ステップ毎に遷移規則によって更新されるのですが、Address Bit だけが周囲にエラーが起きない限りは更新されません。したがって、Gács のルールでは Address Bit の配列が異なる初期状態同士が、「保存」されるようなしかけになっています。

Sim Bit はコロニーによる自己シミュレーションに使われます。図 8 では各セルの Sim Bit を繋ぎ合わせてコロニー単位で見ると、あたかもひとつのセルであるかのように見える様子を描いています。このシミュレートされたセルがコロニーの作業時間単位 U 毎に更新されてゆきます。[1]「自己」シミュレーションと呼ぶ理由は、このシミュレートされたセルの挙動がもとのセルの挙動と全く同じに設計されているからです。

WorkSpace の中を見てみましょう。Workspace の中にはシミュレーションのために Mail Bit と Program Bit と呼ばれる部分があります。

Mail Bit コロニー間の Sim Bit の情報を交換する。ひとつのセルの Sim Bit の情報は、コロニーを越えてその情報を必要とするセルまで伝達される。

Program Bit 他のコロニーやコロニー内部から送られてきた Sim Bit の情報をもとに自分の Sim Bit を書き換える変換規則。Gàcs 独自の言語でインプリメントされている。

単純に、あるセルオートマトンで別のセルオートマトンをシミュレートするのであれば、このようなしかけは必要ありません。コロニーからメタコロニー、メタメタコロニーと連鎖的にシミュレーションを持続さるために必要になるわけです。Program Bit へのプログラムの埋め込みは、Turing Machine における UTM のプログラムの埋め込みに近いと考えればよいでしょう。もちろんひとつのセルの Work Space の中に、全ての必要な Program が内蔵されているわけではなく、コロニー全体にまたがって分散されています。

いま M_0, M_1 をセルオートマトンとして、M_1 は M_0 によってシミュレートされているとします。活動時間のはじめに M_1 の Program Bit は、M_1 の Address とその Address に対応する下位の M_0 のセルの Program Bit と同じになるようにセットする。そうすると M_1 は、自分と全く同じ遷移規則を持つ新たなセルオートマトン M_2 をシミュレートすることができるようになり、同様の工程を M_1 が行えるようにプログラムができていれば自己シミュレーションの連鎖は無限に続くようになるのです。

$$M_0 \rightarrow M_1 \rightarrow M_2 \rightarrow ...$$

5 エラー修正のしくみ

エラー修正は基本的には多数決回路によって行います。ただし Boolean Circuit や Toom のモデルとの違って、多数決のしかたは WorkSpace の中の Flag Bit の状態に応じて変わります。

Flag Bit が **0** であるときは、セルは通常の多数決を行って次の自分の状態を決定します。コロニーの端にいるセルは隣のコロニーのセルも見えるはずですが、通常の状態では隣のコロニーの情報は無視します。一方で Flag Bit が **1** のときは周囲のセルで多数決するのをあきらめて、隣のコロニーのセルの情報を参照してせっせとコピーします。

Flag Bit とはエラーに対する警報の役目をするものだと考えて下さい。Flag Bit は 他のコロニーとの整合性が取れなくなったときに **1** になり、そうでなければ **0** になり、その情報はコロニー内部のセル全てに行き渡ります。したがって、コロニー内部で発生したエラーはコロニーを超えて伝播することなく、隣接するコロニーの情報を利用して修正されるます。このエラー修正は、コロニーの活動単位時間 U 以内で完了し、シミュレートされた上位セルには影響は出ません。

[1]コロニーのコロニー、すなわちメタコロニーによってシミュレートされたセルの単位時間は U^2 になります。

ではもし複数のコロニーわたるエラーが発生した場合にはどうするのでしょうか？その場合にはシミュレーションにされた上位セルの挙動になんらかの影響が出るはずで、上位セルは異常を察知した場合には下位のセルの Flag Bit を 1 にセットすることができます。メタコロニー単位で隣接するメタコロニーの情報をコピーすることになるでしょうが、それでもメタコロニーの単位時間 U^2 内には処理は終了します。

このように自己シミュレーションを利用して Gács のセルオートマトンは 1 次元における境界の問題をクリアしたのです。

6 おわりに

Gács の論文が膨大な理由は動作の全ての局面にわたって、エラーの影響を考慮しなければいけない点にあります。Gács のモデルは想像されるようにかなり大規模です。当面 Gács の成果が実際面で応用されるようなことはないでしょう。

私自身が Gács のモデルをみて一番新鮮さを感じたのは、自己シミュレーションという考え方が Turing 機械計算論では対角線論法の中で否定的に扱われるのに対して、Gács モデルでは積極的な意味で使われているということです。von Neuman の Boolean Circuit にしても Gács のセルオートマトンにしても、基本は再帰的な構成にあるということを見てきましたが、Boolean Circuit ではエラー修正方法を外部から静的に組み込むのに対して、Gács のモデルでは内部から動的に構築するという大きな違いがあります。オートポイエーシス風に言えば、「Gács のセルオートマトンは自らがエラーを定義している」と言えるのではないでしょうか？

参考文献

[1] Peter Gacs, Reliable Cellular Automata with Self-Organization, *J. Stat. Phys.* **103**, 45-267 (2001).

[2] Peter Gacs, *Lectures on Reliable Cellular Automata*, http://citeseer.nj.nec.com/20653.html

[3] Lawrence F. Gray, A Reader's Guide to Gac's "Positive Rates" Paper, *J. Stat. Phys.* **103**, 1-44 (2001).

[4] John von Neuman, Probabilistic logics and the synthesis of reliable organisms from unreliable components, *Automata Studies* (Princeton, NJ.) (1956).