# True Concurrency vs. Nondeterministic Sequential Interleavings in 1-D Cellular Automata

**Predrag Tosic**[*], **Gul Agha**

Open Systems Laboratory, Department of Computer Science
University of Illinois at Urbana-Champaign
**{p-tosic, agha}@cs.uiuc.edu**

## Abstract

**Cellular automata (CA)** are considered an abstract model of *fine-grain parallelism*, in that the elementary operations executed at each node are rather simple and hence comparable to the most elementary operations in the computer hardware. In a classical cellular automaton, all the nodes execute their operations in a truly concurrent manner: the state of node $x_i$ at time step $t+1$ is some simple function of the states of node $x_i$ and a set of its pre-specified neighbors at time $t$. We consider herewith the sequential version of **CA**, or **SCA**, and compare it with the classical, *parallel* (meaning, *truly concurrent*) **CA**. In particular, we show that there are **1-D CA** with very simple node state update rules that cannot be simulated by any comparable **SCA**, irrespective of the node update ordering. We argue that, while the **CA** and **SCA** we consider are very simple, the difference in dynamic behaviors (or, equivalently, the computation properties) are rather fundamental. Hence, perhaps the granularity of basic **CA** operations, insofar as the ability to interpret their concurrent computation via an appropriate nondeterministic sequential interleaving semantics, is not fine enough - namely, we prove that no such sequential interleaving semantics can capture even rather simplistic concurrent **CA** computations. We also share some thoughts on how to extend our early results, and, in particular, motivate introduction and study of **asynchronous cellular automata**.

**Keywords:** *formal methods for RTS, cellular automata, true concurrency, bounded asynchrony, nondeterministic interleaving semantics*

## 1 Introduction & Motivation

While our own brains are massively parallel computing devices, we seem to think and function rather sequentially. In particular, our approach to problem solving is typically highly sequential - whether the problem domain is a simple situation in everyday life or development of a sophisticated algorithm for a highly complex scientific problem. In particular, when designing algorithms or writing computer programs that are purposefully inherently parallel, we prefer to be able to understand such an algorithm or program in the sequential terms, as we are so much more comfortable with the "sequential way of thinking" than with its parallel counterpart. It is not surprising, therefore, that since the very beginning of the design of parallel algorithms and parallel computation formalisms, a great deal of research effort has been devoted to interpreting parallel computation in more familiar, sequential terms. The classical example of this "sequential way of thinking" in computer science is the nondeterministic sequential interleaving semantics of concurrent computation.

In this context, a natural question arises: Given a parallel computing model, can its (logically) concurrent execution always be replaced by such nondeterminism, so that any given concurrent behavior can be faithfully reproduced via an appropriate choice of a sequential interleaving? The answer is *Yes*, provided that we simulate concurrent execution via sequential nondeterministic interleaving at a sufficiently high level of granularity of the basic computational operations. However, how do we tell if the particular level of granularity is *fine enough*, i.e., that the operations at that level can truly be rendered *atomic* for the purposes of capturing concurrency?

Consider the following trivia question from a sophomore parallel programming class: *Find a simple example of two instructions such that, when executed in parallel, they give a result not obtainable from any corresponding sequential execution sequence?*

**Answer:** Assume $x = 0$ initially and consider the programs

$$x \leftarrow x + 1; \ x \leftarrow x + 1$$
$$\text{vs.}$$
$$x \leftarrow x + 1 \ || \ x \leftarrow x + 1$$

Sequentially, one *always* gets the same answer: $x = 2$. In parallel (that is, if the two assignment operations to the same variable $x$ are done concurrently), however, one gets $x = 1$. It appears, therefore, that no sequential ordering of operations can produce parallel computation - at least not at the high level of instructions as above.

The whole "mystery" is resolved if we look at the possible sequential executions of the corresponding machine instructions:

```
LOAD   x, *m      LOAD    x, *m
ADD    x, #1      ADD     x, #1
STORE  x, *m      STORE   x, *m
```

There certainly exists a *sequential interleaving* of the six machine instructions above that leads to *"parallel"* behavior (i.e., the one where, after the code is executed, $x = 1$) - in fact, there are several such permutations of instructions. Thus, it turns out that the example above does not require finer granularity quite yet: high-level "concurrent" computation can be perfectly well understood in terms of the sequential interleaving semantics at the level of machine instructions, which can therefore be considered *atomic*.

---
[*]Contact author

Indeed, if we informally define function $\Phi(P)$ as the *meaning of program P*, then the example above only shows that, for $S_1 = S_2 = (x \leftarrow x + 1)$,

(1)    $\Phi(S_1 \| S_2) \neq \Phi(S_1; S_2) \cup \Phi(S_2; S_1)$

where "$\|$" stands for parallel, and "$;$" for sequential composition of instructions (or programs), respectively. The good news in this example, as it turns out, is that it is the case that

(2)    $\Phi(S_1 \| S_2) = \Phi(S_1; S_2) \cup \Phi(S_2; S_1) \cup \Phi(S_1) \cup \Phi(S_2)$

and no finer granularity is necessary to model $\Phi(S_1 \| S_2)$, assuming that, in the interleaving semantics, we allow certain instructions not to be executed at all.

However, one can construct more elaborate examples where (2) (or any of its obvious extensions) do not hold. The only way to model $\Phi(S_1 \| S_2)$ via nondeterministic sequential interleavings in such cases would then be to find a finer level of granularity, i.e., to reconsider at what level can operations be considered *atomic*, so that nondeterministic interleavings of such basic operations are guaranteed to be able to capture the "concurrent" behavior. That is, sometimes *refining the granularity of operations* so that the semantics of nondeterministic sequential interleavings suffices for interpreting concurrency, becomes a **necessity**. In the example above, this granularity refinement was from $S_i$ in high-level language down to machine *loads* and *stores*.

# 2   Cellular Automata and Types of Their Configurations

We introduce **CA** by first considering **Finite State Machines (FSMs)** such as **Deterministic Finite Automata (DFA)**. **FSMs** have finitely many states, and are capable of reading input signals coming from the outside; let's assume these signals are 0s and 1s. The machine is initially in some starting state; upon reading each input signal (a single binary symbol, in the standard **DFA** case), the machine changes its state according to a pre-defined and fixed rule. In particular, the entire memory of the system is contained in what "current state" the machine is in, and nothing else about previously processed inputs is remembered. Hence, the probabilistic generalization of deterministic **FSMs** leads to (discrete) Markov chains. Second, there is no way for a **FSM** to overwrite, or in any other way affect the input data stream. Thus *individual* **FSMs** are computational devices of rather limited power.

Now let us consider many of such **FSMs**, all identical to one another, that are lined up together in some regular fashion, e.g. on a straight line or a regular 2-D grid, so that each single "node" is connected to its immediate neighbors. Let's also eliminate any external sources of input streams to the individual machines that are the nodes in our grid, and let the current values of any given node's neighbors be that node's only "input data". If we then specify the set of the values held in each node (typically, this set is $\{0, 1\}$), and we also identify this set of values with the set of the node's *internal states*, we arrive at an informal definition of a cellular automaton. To summarize, a **CA** is a finite or infinite regular grid in one-, two- or higher-dimensional space, where each node in the grid is a particularly simple **FSM**, with only two possible internal states, 0 and 1, and whose input data at each time step are the corresponding internal states of its neighbors. All the nodes execute the **FSM** computation in unison, i.e. (logically) simultaneously. Thus, **CA** can be viewed as a parallel model where perfect synchrony is assumed. We note that infinite **CA** are capable of universal (Turing) computation, and, moreover, are actually strictly more powerful than classical Turing machines (see, e.g., [GARZ]).

More formally, we use the following definition of a **CA** ([GARZ]):

**Definition 1:**    *Cellular Space* $\Gamma$ is an ordered pair $(G, Q)$ where: - $G$ is a regular graph (finite or infinite); and - $Q$ is a finite set of states that has at least two elements and includes a special *quiescent state* 0.

**Definition 2:**    *Cellular Automaton* $A$ is an ordered triple $(\Gamma, N, M)$ where:
- $\Gamma$ is a *cellular space*;
- $N$ is a *fundamental neighborhood*;
- $M$ is a *finite state machine* such that the input alphabet of $M$ is $Q^{|N|}$, and the local transition function (update rule) for each node is of the form $\delta : Q^{|N|+1} \to Q$ for **CA with memory**, and $\delta : Q^{|N|} \to Q$ for **memoryless CA**.

Local transition rule $\delta$ specifies how each node updates, based on its current value and that of its neighbors in $N$. By composing local transition rules for all nodes together, we obtain *global map* on the set of all **CA** configurations. Herein, we only consider **Boolean CA**: the only possible states of any node are 0 and 1.

As for any other kind of dynamical systems, we can define the fundamental, qualitatively distinct types of configurations that a cellular automaton can find itself in.

**Definition 3:**    A *fixed point (FP)* is a configuration in the phase space of a **CA** such that, once **CA** reaches this configuration, it stays there forever. A *cycle configuration (CC)* is a state that, if once reached, will be revisited infinitely often with a (finite) period of 2 or greater. A *transient configuration (TC)* is a state that, once reached, is never going to be revisited again.

In particular, a FP is a special, degenerate case of CC with period 1. Due to deterministic evolution, any configuration of a **CA** is either a FP, a proper CC, or a TC.

Our main results outlined in the next section compare and contrast classical one-dimensional **CA**, where all the nodes update in parallel, and their sequential counterparts, where nodes update one at the time, that is, sequentially.

# 3   1-D CA vs. SCA Comparison and Contrast: The Existence of Cycles

A **1-D cellular automaton of radius** $r$ is a **CA** defined over a one-dimensional string of nodes, such that each node's next state depends on the current states of its neighbors to the left and right that are no more than $r$ nodes away. (and, in case of **CA** *with memory*, on the current state of itself). In case of a **Boolean CA** *with memory*, therefore, each node's next state depends on $2r + 1$ input bits, while in *memoryless case*, the local update rule is a function of $2r$ input bits. The string of nodes can be a finite line graph, a ring (corresponding to "circular boundary conditions"), or, in the most common case one finds in the literature, the cellular space is a two-way infinite string.

We compare and contrast the qualitative properties of configurations spaces, and therefore dynamics or possible computations, of the classical, truly concurrent **CA** vs. their sequential counterparts. Sequential **CA** and their generalizations to non-regular graphs have been studied, e.g., in the context of a formal theory of computer simulation (see, e.g., [BARE]).

The results below hold for two-way infinite **1-D CA**, as well as for finite **CA** with sufficiently many nodes and circular boundary conditions (i.e., for **CA** whose cellular spaces are finite rings). We remark that the "inputs" that a given node of a **CA** uses to compute its next state, are

the current states of that node's neighbors and of the node itself.

**Lemma 1:**
(i) 1-D classical (i.e., parallel) **CA** with $r = 1$ and the **MAJORITY** update rule has (finite) cycles in the phase space ($PS$).

(ii) **1-D Sequential CA** with $r = 1$ and **MAJORITY** update rule do not have any (finite) cycles in the phase space, *irrespective* of the sequential node update order $\rho$.

**Remark:** In case of infinite sequential **CA** as in the Lemma above, a nontrivial cycle configuration does not exist even in the limit. We also note that $\rho$ is an arbitrary sequence of **CA** nodes' indices, not necessarily a (finite or infinite) permutation.

**Proof:**
To show (i), we exhibit an actual two-cycle. Consider either an infinite **1-D CA**, or a finite one, with circular boundary conditions and an even number of nodes, $2n$. Then the configurations $(10)^\star$ and $(01)^\star$ in the infinite case $((10)^n$ and $(01)^n$ in the finite ring case) form a 2-cycle.

To prove (ii), we must show that no cycle is ever possible, irrespective of the starting configuration. We consider all possible 1-neighborhoods (there are eight of them: 000, 001, ..., 111), and show that, locally, none of them can be cyclic yet not fixed. The case analysis is simple: 000 and 111 are stable (fixed) sub-configurations. Configuration 010, after a single node update, can either stay fixed, or else evolve into any of $\{000, 110, 011\}$; since we are only interested in non-FPs, in the latter case, one can readily show by induction that, after any number of steps, the only additional sub-configuration that can be reached is 111, i.e., assuming 010 is not fixed, $010 \rightarrow^\star \{000, 110, 011, 111\}$. However, $010 \notin \{000, 110, 011, 111\}$. By symmetry, similar analysis holds for sub-configuration 101. On the other hand, 110 and 011 either remain fixed, or else at some time step $t$ evolve to 111, which is a fixed point. Similar analysis applies to 001 and 100. Hence, no local neighborhood $x_1 x_2 x_3$, once changed, can ever "come back". Therefore, there are no proper cycles in Sequential 1-D **CA** with $r = 1$ and $\delta =$ **MAJORITY**.[1]

Part (ii) of the Lemma above can be generalized: even if we consider local update rules $\delta$ other than the **MAJORITY** rule, yet restrict $\delta$ to *monotone symmetric (Boolean) functions* of the input bits, such sequential **CA** still do not have any proper cycles.

**Theorem 1:**
For any **Monotone Symmetric Boolean 1-D Sequential CA** $A$ with $r = 1$, and any sequential update order $\rho$, the phase space of $A$ is cycle-free.

**Remark:** There are, really, only *five* Monotone Symmetric Boolean (**MSB**) functions on three inputs, two of which utterly trivial (the constant functions 0 and 1). It then follows that, for all of these sequential **CA**, the corresponding phase spaces, viewed as directed graphs, are forests of quasi-trees (where each quasi-tree has a self-loop at the root, and the edges are directed from leaves towards the root).

We now state similar results to those in Lemma 1 and Theorem 1 for **1-D CA** with radii $r = 2$ and $r = 3$, respectively. The next Lemma should not come as a surprise:

**Lemma 2:**
(i) **1-D CA** with $r \in \{2, 3\}$ and with the **MAJORITY** node update rule have (finite) cycles in the phase space.

(ii) A **1-D CA** with **MAJORITY** node update rule, $r = 2$ and any sequential order on node updates has a cycle-free phase space.

**Proof:**
(i) For $r = 2$, consider configurations $(1100)^\star$ and $(0011)^\star$; clearly they form a 2-cycle for (parallel) **CA**. In case of $r = 3$, and for all odd $r$, the two configurations from **Lemma 1 (i)** are still forming a 2-cycle; there are other 2-cycles in configuration spaces of such **CA**, as well.

(ii) Proof is similar to that of **Lemma 1 (i)**, except that now there are $2^5 = 32$ fundamental neighborhoods of the form $x_1 ... x_5$ to consider; it is worth observing that, for $r = 2$, sub-configurations 000 and 111 are stable - this observation considerably simplifies the tedious case analysis.

We do not know at this time, if **1-D SCA** with **MAJORITY** node update rule and $r \geq 3$ may have proper cycles. As for the (parallel) **CA**, it is easy to generalize the above results on the existence of cycles:

**Corollary 1** (generalizing Lemmata 1 and 2, part (i)):
For all $r \geq 1$, there exists a monotone symmetric Boolean CA $A$ such that $A$ has (finite) cycles in the phase space. Namely: $\forall r \geq 1$, (parallel) CA with $\delta =$ **MAJORITY** have finite temporal cycles in the configuration space.

# 4 Discussion and Future Directions

The results in §3 show that, even for the very simplest cellular automata, nondeterministic interleavings dramatically fail to capture concurrency. It is not surprising that one can find a (parallel) **CA** such that no sequential **CA** with the same underlying cellular space and the same node update rule can produce identical computation. However, we find it rather interesting that very profound differences (a possibility of looping vs. the guaranteed convergence to a fixed point) can be observed in case of extremely simple parallel and sequential **CA** - viz., one-dimensional automata with small $r$ and simple threshold functions as the node update rules.

There are several possible directions to pursue in order to generalize the results herein. One of these directions, already explored (and whose results are reported elsewhere), is to consider CA-like finite automata defined over *arbitrary* (rather than only regular) graphs. In that context, results similar to those in §3 have been obtained for parallel and sequential automata defined over arbitrary *bipartite graphs*: such graph automata, if the nodes are updated concurrently, also may contain nontrivial cycles even in case of the simplest node update rules such as **MAJORITY**, yet no cycles are possible if the nodes are updated sequentially.

Another direction for further investigation is to consider other models of (a)synchrony in CA. Classical (that is, concurrent) **CA** can be viewed as a class of computational models of *bounded parallelism*: if nodes $x$ and $y$ are at distance $k$ (i.e., $k$ nodes apart from each other), and the radius of the **CA** is $r$, then any change in the state of $y$ can affect the state of $x$ no sooner, but also (and perhaps more importantly) no later than after about $\frac{k}{r}$ (parallel node update) computational steps. As the nodes all update "in sink", locally a classical **CA** is a synchronous concurrent system. However, globally, i.e., if one is interested in the interactions of nodes that are at a distance greater than $r$ apart, we argue that the classical **CA** and their various graph automata

---

[1] An astute reader will notice that the above case analysis in the proof of (ii) can be simplified if one observes that, for $r = 1$, sub-configurations 11 and 00 are always stable with respect to the **MAJORITY** function, irrespective of the left or right neighbors, or the updating order.

extensions[2] can be readily viewed as a class of models of *bounded asynchrony*. It would be interesting, therefore, to consider *asynchronous cellular (or graph) automata*, where not all the nodes necessarily update in sink.

The simplest **asynchronous CA** we have considered are just like the classical (synchronous) **CA**, except that a given node update function, instead of current, may be using old values of some of its inputs (that is, neighbors' states). That is, whereas in the classical case $x_i^{t+1} = f(x_i^t, x_{i_1}^t, ..., x_{i_{2r}}^t)$, in asynchronous case, $x_i^{t+1} = f(x_i^t, x_{i_1}^{t_1}, ..., x_{i_{2r}}^{t_{2r}})$, where each $t_j \in \{0, ..., t\}$, $j = 1, 2, ..., 2r$. Thus such asynchronous **CA** can be seen as *models of unbounded asynchrony*. This asynchrony can also be readily interpreted in nondeterministic terms: at each time step, the node updates using its own current value, and also using the current or some of the past values of its neighbors. In other words, each node has its own clock and, from the viewpoint of what are the current inputs to any given node update function, there is a natural nondeterministic interpretation of the fact that the nodes need not be in sink.

Many interesting questions arise in this context. One is, what kinds of phase-space properties remain invariant under this kind of nondeterminism? Given a $(\Gamma, N, M)$, it can readily be shown that fixed points are invariant with respect to the node update ordering in synchronous sequential **CA**, and, moreover, are the same for the corresponding parallel **CA**. On the other hand, as our results in §3 indicate, neither cycle configurations nor transient configurations are invariant with respect to whether the nodes are updated sequentially or concurrently (and, in case of the former, in what order). It can be readily observed that, indeed, FPs are invariant for asynchronous **CA** and graph automata, and therefore the earlier results in [BHM+] for (synchronous) sequential and concurrent graph automata are just special cases of this result. We record this easy to prove result below:

**Theorem 2:**
Given an arbitrary asynchronous **CA** (or graph automaton), any fixed point configuration is invariant with respect to the node update orders and the delays due to asynchrony.

In addition to studying invariants under different assumptions on asynchrony and concurrency, we also consider qualitative comparison-and-contrast of the asynchronous **CA** that we propose, and the classical **CA**; such analysis could shed light on detecting computational behaviors that are solely due to (our abstracted version of) network delays.

More generally, asynchronous **CA**, i.e., various nondeterministic choices for a given **CA** that are due to asynchrony, can be shown to subsume all possible behaviors of classical and sequential **CA** with the same corresponding $(\Gamma, N, M)$. In particular, the nondeterminism that arises from (unbounded) asynchrony subsumes the nondeterminism of a kind studied in §3; but the question arises, exactly how much more expressive and powerful the former nondeterministic interleaving semantics really is. In addition to being more realistic, asynchronous **CA** are also likely promising an interesting way of generalizing classical **CA** and, in particular, exploring various implications of the unrealistic yet convenient synchrony or bounded asynchrony assumptions at an abstract level.

# 5   Conclusions

In this short paper, we outline early steps of studying cellular automata when the unrealistic assumption on true concurrency and instantaneous massive parallelism are relaxed. Motivated by the well-known notion of nondeterministic interleaving sequential semantics of concurrency, we apply this concept to parallel and sequential **CA**, and show that, even in very simplistic cases, this sequential semantics fails to capture true concurrency of classical **CA**. One lesson is that, simple as they may be, the basic operations (node updates) in classical **CA** cannot always be considered atomic; that is, the fine-grain parallelism of **CA** turns out not to always be fine enough. Motivated by these early results on sequential and parallel **CA** (see §3), we are considering various extensions. The central idea is to introduce a class of **asynchronous CA**, and formally study their properties. This would hopefully eventually yield some significant insights into the fundamental issues related to bounded vs. unbounded asynchrony, formal semantics for concurrent and distributed computation, and abstract formalisms for qualitative study of RTSs. We also feel that various extensions of the basic **CA** model provide a simple, elegant and useful framework for studying various qualitative properties of various distributed and real-time systems at an abstract, yet formal and rigorous mathematical level.

### Bibliography
[BARE] Barrett, C. and Reidys, C. (1999) "Elements of a theory of computer simulation I: sequential CA over random graphs", Applied Math. & Computation, vol. 98

[BHM+] Barrett, C., Hunt, H., Marathe, M., Ravi S. S., Rosenkrantz, D., Stearns, R. and Tosic, P. (2001) "Gardens of Eden and Fixed Points in Sequential Dynamical Systems", Discrete Math. & Theoretical Comp. Sci. Proc. AA (DM-CCG)

[BMR1] Barrett, C., Mortveit, H. and Reidys, C. (2000) "Elements of a theory of computer simulation II: sequential dynamical systems", Applied Math. & Computation, vol. 107/2-3

[BMR2] Barrett, C., Mortveit, H. and Reidys, C. (2000) "Elements of a theory of computer simulation III: equivalence of sequential dynamical systems", Applied Math. & Computation

[GARZ] Garzon, M. (1995) "Models of Massive Parallelism: Analysis of Cellular Automata and Neural Networks", Springer

[MIL1] Milner, R. (1980) "A Calculus of Communicating Systems", Lecture Notes in CS

[MIL2] Milner, R. (1983) "Calculi for synchrony and asynchrony", Theoretical CS 25

[MIL3] Milner, R. (1989) "Communication and Concurrency", Prentice-Hall

[SUTN] Sutner, K. (1998) "Computation theory of cellular automata", MFCS98 Satellite Workshop on CA, Brno, Czech Rep.

[WOL1] Wolfram, S. (1985) "Twenty problems in the theory of CA", Physica Scripta 9

[WOL2] Wolfram, S. (ed.) (1986) "Theory and applications of CA", World Sci., Singapore

---

[2] Two classes of such graph automata, viz., sequential and synchronous dynamical systems (SDSs and SyDSs, respectively), and their various phase space properties, have been extensively studied; see, e.g., [BHM+], [BMR1], [BMR2], [REID] and references therein.