Bounded Asynchrony: Concurrency for Modeling Cell-Cell Interactions ⋆

Jasmin Fisher[1], Thomas A. Henzinger[2], Maria Mateescu[2], and Nir Piterman[3]

[1] Microsoft Research, Cambridge UK
[2] EPFL, Switzerland
[3] Imperial College London, UK

Abstract. We introduce *bounded asynchrony*, a notion of concurrency tailored to the modeling of biological cell-cell interactions. Bounded asynchrony is the result of a scheduler that bounds the number of steps that one process gets ahead of other processes; this allows the components of a system to move independently while keeping them coupled. Bounded asynchrony accurately reproduces the experimental observations made about certain cell-cell interactions: its constrained nondeterminism captures the variability observed in cells that, although equally potent, assume distinct fates. Real-life cells are not "scheduled", but we show that distributed real-time behavior can lead to component interactions that are observationally equivalent to bounded asynchrony; this provides a possible mechanistic explanation for the phenomena observed during cell fate specification.

We use model checking to determine cell fates. The nondeterminism of bounded asynchrony causes state explosion during model checking, but partial-order methods are not directly applicable. We present a new algorithm that reduces the number of states that need to be explored: our optimization takes advantage of the bounded-asynchronous progress and the spatially local interactions of components that model cells. We compare our own communication-based reduction with partial-order reduction (on a restricted form of bounded asynchrony) and experiments illustrate that our algorithm leads to significant savings.

1 Introduction

Computational modeling of biological systems is becoming increasingly important in efforts to better understand complex biological behaviors. In recent years, formal methods have been used to construct and analyze such biological models. The approach, dubbed "executable biology" [10], is becoming increasingly popular. Various formalisms are putting the executable biology framework into practice. For example, Petri-nets [3, 7], process calculi [22, 15], interacting state-machines [9, 11], and hybrid automata [13, 2]. In many cases, the analysis of these models includes reachability analysis and model checking in addition to traditional simulations.

This paper focuses on interacting state-machines as a tool for biological modeling [18, 8, 19, 12, 23, 9, 11]. This approach has recently led to various biological discoveries, and modeling works that were done using this approach have appeared in high

⋆ Supported in part by the Swiss National Science Foundation (grant 205321-111840).

impact biological journals [12, 9, 11]. These are discrete, state-based models that are used as high-level abstractions of biological systems' behavior.

When using interacting state-machine models to describe a biological behavior, we are facing the question of how to compose its components. We find that the two standard notions of concurrency (in this context), synchrony and asynchrony, are either too constrained or too loose when modeling certain biological behaviors such as cell-cell interactions. When we try to model cell-cell interactions, we find that synchronous composition is too rigid, making it impossible to break the symmetry between processes without the introduction of additional artificial mechanisms. On the other hand, asynchronous composition introduces a difficulty in deciding when to stop waiting for a signal that may never arrive, again requiring artificial mechanisms.[4]

Biological motivation.

We further explain why the standard notions of concurrency may be inappropriate for modeling certain biological processes. We give a model representing very abstractly a race between two processes in adjacent cells that assume two different cell fates. The fate a cell chooses depends on two proteins, denoted *pathway* and *signal*, below. The pathway encourages the cell to adopt fate1 while the signal encourages the cell to adopt fate2. In the process we are interested in, pathway starts increasing slowly. When pathway reaches a certain level, it forces the cell to adopt fate1. At the same time, pathway encourages the signal in neighbor cells to increase and inhibits the pathway in the neighbor cell. The signal starts in some low level and if not encouraged goes down and vanishes. If, however, it is encouraged, it goes up, inhibiting the pathway in the same cell, and causing the cell to adopt fate2. A simple model reproducing this behavior is given in Fig. 1.

We are interested in three behaviors. First, when a cell is run in isolation, the pathway should prevail and the cell should assume fate1. Second, when two cells run in parallel either of them can get fate1 and the other fate2. There are also rare cases where both cells assume fate1. Third, when one of the cells gets an external boost to the pathway it is always the case that this cell adopts fate1 and the other fate2.

Already this simplified model explains the problems with the normal notions of concurrency. In order to allow for the second behavior we have to break the symmetry between the cells. This suggests that some form of asynchrony is appropriate. The combination of the first and third behaviors shows that the asynchronicity has to be bounded. Indeed, in an asynchronous setting a process cannot distinguish between the case that it is alone and the case that the scheduler chooses it over other processes for a long time.

Although very simple, this model is akin to many biological processes in different species. For example, a similar process occurs during the formation of the wing of the Drosophila fruit fly [13]. Ghosh and Tomlin's work provides a detailed model

[4] We treat here biological processes as computer processes. For example, when we say 'waiting', 'message', or 'decide' we relate to biological processes that take time to complete, and if allowed to continue undisturbed may lead to irreversible consequences. Thus, as long as the process is going on the system 'waits', and if the process is not disturbed ('does not receive a message'), it 'decides'.

```
var pathway,signal:{0..4};

pathway_atom
init
[] true -> path := 1;
update
[] (0<path<4) & no_input & next(signal)<4 -> path := path+1;
[] (0<path<4) & input & next(signal)<4 -> path := 4;
[] (0<path<4) & next(signal)=4 -> path := 0;

signal_atom
init
[] true -> signal := 3;
update
[] neighborpath=4 & signal>0 -> signal := 4;
[] neighborpath<4 & path=4 -> signal := 0;
[] neighborpath<4 & path<4 & 0<signal<4 -> signal := signal-1;
```

Fig. 1. Program for abstract model.

(using hybrid automata) of this process. The formation of the C. elegans vulva also includes a similar process [11]. Our model of C. elegans vulval development uses the notion of bounded asynchrony. Using bounded asynchrony we separate the modeling environment from the model itself and suggest biological insights that were validated experimentally [11].

Formal modeling: bounded asynchrony.

For this reason, we introduce a notion of *bounded asynchrony* into our biological models, which allows components of a biological system to proceed approximately along the same time-line. In order to implement bounded asynchrony, we associate a rate with every process. The rate determines the time t that the process takes to complete an action. A process that works according to rate t performs, in the long run, one action every tth round. This way, processes that work according to the same rate work more or less concurrently, and are always at the same stage of computation, however, the action itself can be taken first by either one of the processes or concurrently, and the order may change from round to round[5]. Other notions of bounded asynchrony either permit processes to 'drift apart', allowing one process to take arbitrarily more actions than another process, or do not generalize naturally to processes working according to different rates.

Having the above mentioned example in mind, we define the notion of bounded asynchrony by introducing an explicit scheduler that instructs each of the cells when it is allowed to move. Thus, our system is in fact a synchronous system with a nondeterministic scheduler instructing which processes to move when. We find this notion of

[5] We note that this process is not memoryless, making continuous time Markov chains inappropriate. This issue is discussed further below.

bounded asynchrony consistent with the observations made in cell-cell interactions. As explained, asynchrony is essential in order to break the symmetry between cells (processes). It is important to separate the biological mechanism from the synchronization mechanism, otherwise the model seems removed from the biology. On the other hand, much like in distributed protocols, a process has to know when to give up on waiting for messages that do not arrive. With classical asynchrony this is impossible and we are forced to add some synchronizing mechanism. Again, in the context of biology, such a mechanism should be presented in terms of the modeling environment. When introducing bounded asynchrony both problems are solved. The asynchrony breaks the symmetry and the bound allows processes to decide when to stop waiting. In addition, the asynchrony introduces limited nondeterminism that captures the diversity of results often observed in biology.

Possible mechanistic explanation: real time.

In some cases, biological systems allow central synchronization. For example, during animal development, it may happen that several cells are arrested in some state until some external signal tells all of them to advance. However, these synchronization mechanisms operate on a larger scale and over time periods that are much longer than the events described by our model. Thus, we do not believe that there is a centralized scheduler that instructs the processes when to move. The behaviors we describe are observed in practice, suggesting that there is some mechanism that actually makes the system work this way. This mechanism has to be distributed between the cells. We show that bounded asynchrony arises as a natural abstraction of a specific type of clocked transition systems, where each component has an internal clock. This suggests that similar ideas may be used for the abstraction of certain types of real time systems. Of less importance here, it also may be related to the actual mechanism that creates the emergent property of bounded asynchrony.

Model checking: scheduler optimization.

The scheduler we introduce to define bounded asynchrony consists of adding variables that memorize which of the processes has already performed an action in the current round. When we come to analyze such a system we find that, much like in asynchronous systems, many different choices of the scheduler lead to the same states. Motivated by partial-order reduction [6], we show that in some cases only part of the interleavings need to be explored. Specifically, our method applies in configurations of the system where communication is locally restricted. In such cases, we can suggest alternative schedulers that explore only a fraction of the possible interleavings, however, explore all possible computations of the system. We also compare our techniques with partial-order reduction in a restricted setting with no concurrent moves. Experimental evaluation shows that our techniques lead to significant improvement. We are not familiar with works that analyze the structure of communication in a specific concurrent system and use this structure to improve model checking.

Related (and unrelated) models.

The comparison of such abstract models with the more detailed differential equations or stochastic process calculi models is a fascinating subject, however, this is not the focus of this paper. Here, we assume that both approaches can suggest helpful insights to biology. We are also not interested in a particular biological model but rather in advancing the computer science theory supporting the construction of abstract biological models.

There are mainly two approaches to handle concurrency in abstract biological models. One prevalent approach is to create a continuous time Markov chain (CTMC). This approach is usually used with models that aim to capture molecular interactions [14, 22]. Then, the set of enabled reactions compete according to a continuous probability rate (usually, the χ-distribution). Once one reaction has occurred, a new set of enabled reactions is computed, and the process repeats. This kind of model requires exact quantitative data regarding number of molecules and reaction rates. Such accurate data is sometimes hard to obtain; indeed, even the data as to exactly which molecules are involved in the process may be missing (as is the case in the *C. elegans* model). Our models are very far from the molecular level, they are very abstract, and scheduler choices are made on the cellular level. When considering processes abstractly the scheduling is no longer memoryless, making CTMCs inappropriate. For example, consider a CTMC obtained from our model in Fig. 1 by setting two cells in motion according to the same rate. Consider the experiment where one of the cells is getting a boost to its pathway. The probability of the other cell performing 4 consecutive actions (which would lead to it getting fate1) is $\frac{1}{16}$, while this cannot occur in the real system. In addition, the probability of both cells assuming fate1 is 0, as the cells cannot move simultaneously.

A different approach, common in *Boolean networks* [20, 4, 5], is to use asynchrony between the substances. Again, this approach is usually applied to models that aim to capture molecular interactions, however, in an abstract way. Asynchronous updates of the different components is used as an over-approximation of the actual updates. If the system satisfies its requirements under asynchronous composition, it clearly satisfies them under more restricted compositions. We note, however, that these models are used primarily to analyze the steady-state behavior of models (i.e., loops that have no outgoing edges). As asynchrony over-approximates the required composition, such steady-state attractors are attractors also in more restricted compositions, justifying this kind of analysis. For our needs, we find unbounded asynchrony inappropriate.

Bounded asynchrony is in a sense the dual of GALS (globally-asynchronous-locally-synchronous): it represents systems that look globally, viewed at a coarse time granularity, essentially synchronous, while they behave locally asynchronous, at a finer time granularity. Efficient implementations of synchronous embedded architectures also fall into this category. For example, time-triggered languages such as Giotto [16] have a synchronous semantics, yet may be implemented using a variety of different scheduling and communication protocols.

2 Bounded Asynchrony

In this section we define the notion of bounded asynchrony. We first define transition systems and then proceed to the definition of bounded asynchrony.

2.1 Transition Systems

A *transition system* (TS) $\mathcal{D} = \langle V, W, \Theta, \rho \rangle$ consists of the following components.

- $V = \{u_1, \ldots, u_n\}$: A finite set of typed *state variables* over finite domains. We define a *state* s to be a type-consistent interpretation of V, assigning to each variable $u \in V$ a value $s[u]$ in its domain. We denote by Σ the set of all states. For an assertion φ, we say that s is a φ-state if $s \models \varphi$.
- $W \subseteq V$: A set of *owned* variables. These are the variables that only $\mathcal{D}$ may change. The set W includes the Boolean *scheduling variable* a.
- Θ : The *initial condition*. This is an assertion characterizing all the initial states of the TS. A state is called *initial* if it satisfies Θ.
- ρ: A *transition relation*. This is an assertion $\rho(V, V')$, relating a state $s \in \Sigma$ to its $\mathcal{D}$-successor $s' \in \Sigma$ by referring to both unprimed and primed versions of the state variables. The transition relation $\rho(V, V')$ identifies state s' as a *$\mathcal{D}$-successor* of state s if $(s, s') \models \rho(V, V')$. The transition relation ρ has the form $(a{\neq}a' \wedge \rho') \vee (W{=}W')$, where a is the scheduling variable. In what follows we restrict our attention to systems that use a scheduling variable.

A *run* of $\mathcal{D}$ is a sequence of states $\sigma : s_0, s_1, ...$, satisfying the requirements of (a) *Initiality:* s_0 is initial, i.e., $s_0 \models \Theta$; (b) *Consecution:* for every $j \geq 0$, the state s_{j+1} is a $\mathcal{D}$-successor of the state s_j. We denote by *runs*$(\mathcal{D})$ the set of runs of $\mathcal{D}$. We can divide the run to transitions where $\mathcal{D}$ stutters (i.e., a and all variables in W do not change) and where $\mathcal{D}$ moves (i.e., a flips its value and variables in W may change).

Given systems $\mathcal{D}_1 : \langle V_1, W_1, \Theta_1, \rho_1 \rangle$ and $\mathcal{D}_2 : \langle V_2, W_2, \Theta_2, \rho_2 \rangle$ such that $W_1 \cap W_2 = \emptyset$, the *parallel composition*, denoted by $\mathcal{D}_1 \parallel \mathcal{D}_2$, is the TS $\langle V, W, \Theta, \rho \rangle$ where $V = V_1 \cup V_2$, $W = W_1 \cup W_2 \cup \{a\}$, $\Theta = \Theta_1 \wedge \Theta_2$, and $\rho = \rho_1 \wedge \rho_2 \wedge \rho'$, the variable a is the scheduling variable of $\mathcal{D}_1 \parallel \mathcal{D}_2$ and ρ' is as follows. [6]

$$\rho' = (a \neq a') \iff [(a_1 \neq a_1') \vee (a_2 \neq a_2')]$$

For more details, we refer the reader to [21].

The *projection* of a state s on a set $V' \subseteq V$, denoted $s\Downarrow_{V'}$, is the interpretation of the variables in V' according to their values in s. Projection is generalized to sequences of states and to sets of sequences of states in the natural way.

2.2 Explicit Scheduler

We define bounded asynchrony by supplying an explicit scheduler that lets all processes proceed asynchronously, however, does not permit any process to proceed faster than other processes. Intuitively, the system has one macro-step in which each of the processes performs one micro-step (or sometimes none), keeping all processes together (regarding the number of actions). The order of actions between the subprocesses is completely non-deterministic. Thus, some of the processes may move together and some one after the other. We start with a scheduler that allows all processes to proceed according to the same rate. We then explain how to generalize to a scheduler that implements bounded asynchrony between processes with different rates.

[6] Notice that, in the case that $\mathcal{D}_1$ and $\mathcal{D}_2$ have stutter transitions, this composition is neither synchronous nor asynchronous in the classical sense.

We start by considering a set of processes all working according to the same rate (without loss of generality the rate is 1). In this case, the resulting behavior is that every process does one micro-step in every macro-step of the system. Namely, we can choose a subset of the processes, let them take a move, then continue with the remaining processes until completing one macro-step. We create a TS that schedules actions accordingly. The scheduler has a Boolean variable b_i associated with every process P_i. A move of P_i is forced when b_i changes from false to true. Once all b_is are set to true, they are all set concurrently to false (and no process moves).

More formally, consider n TSs $P_1, \ldots, P_n$. For $1 \leq i \leq n$, let a_i be the scheduling variable of P_i and let $(\rho_i \wedge a_i \neq a_i') \vee (W_i = W_i')$ be the transition relation of P_i. We define a scheduler $S = \langle V, W, \Theta, \rho \rangle$, where $V = W = \{b_1, \ldots, b_n\}$ and b_i is Boolean for all $1 \leq i \leq n$, $\Theta = \bigwedge_{i=1}^{n} \overline{b_i}$, and ρ is defined as follows:

$$\rho = \left(\Omega \rightarrow \bigwedge_{i=1}^{n} (b_i \rightarrow b_i') \quad \wedge \quad \overline{\Omega} \rightarrow \bigwedge_{i=1}^{n} \overline{b_i'} \right) \tag{1}$$

Where $\Omega = \bigvee_{i=1}^{n} \overline{b_i}$ denotes the assertion that at least one variable b_i is still false.

The *bounded asynchronous parallel composition* of $P_1, \ldots, P_n$ according to the rate 1, denoted $P_1^1 \parallel_{ba} \cdots \parallel_{ba} P_n^1$, is $S \parallel P_1 \parallel \cdots \parallel P_n$ with the following additional conjunct added to the transition:

$$\bigwedge_{i=1}^{n} (a_i \neq a_i' \iff (\overline{b_i} \wedge b_i')) \tag{2}$$

Thus, the scheduling variable of P_i is forced to change when b_i is set to true.

We consider now the more general case of processes working with general rates. In this case, we use the same system of Boolean variables but in addition have a counter that counts the number of steps. A process is allowed to make a move only when its rate divides the value of the counter. More formally, let the rates of $P_1, \ldots, P_n$ be $t_1, \ldots, t_n$. For $1 \leq i \leq n$, let a_i be the scheduling variable of P_i and let $(\rho_i \wedge a_i \neq a_i') \vee (W_i = W_i')$ be the transition relation of P_i. We define a scheduler $S = \langle V, W, \Theta, \rho \rangle$ with the following components:

- $V = W = \{b_1, \ldots, b_n, c\}$. Forall i we have b_i is Boolean, and c ranges over $\{1, \ldots, lcm(t_1, \ldots, t_n)\}$, where lcm is the least common multiplier.
- $\Theta = (c{=}1) \wedge \bigwedge_{i=1}^{n} \overline{b_i}$.
- Let $\Omega = \bigvee_{i=1}^{n} (\overline{b_i} \wedge (c \bmod t_i{=}0))$ denote the assertion that at least one variable b_i for which the rate t_i divides the counter is still false.

$$\rho = (\Omega \rightarrow \bigwedge_{i=1}^{n} (b_i \rightarrow b_i') \wedge (c{=}c')) \wedge \begin{array}{l} \left(\overline{\Omega} \rightarrow \bigwedge_{i=1}^{n} (\overline{b_i'} \wedge (c'{=}c \oplus 1)) \right) \\ \left(\bigwedge_{i=1}^{n} ((\overline{b_i} \wedge (c \bmod t_i{\neq}0)) \rightarrow \overline{b_i}') \right) \end{array} \wedge \tag{3}$$

The *bounded asynchronous parallel composition* of $P_1, \ldots, P_n$ according to rates $t_1, \ldots, t_n$, denoted $P_1^{t_1} \parallel_{ba} \cdots \parallel_{ba} P_n^{t_n}$, is $S \parallel P_1 \parallel \cdots \parallel P_n$ with the the conjunct in Equation (2) added to the transition.

We note that there are many possible ways to implement this restriction of the possible interleavings between processes. Essentially, they all boil down to counting the number of moves made by each process and allowing / disallowing processes to move according to the values of counters.

3 Model Checking

Partial Order Reduction (POR) [6] is a technique that takes advantage of the fact that in asynchronous systems many interleavings lead to the same results. It does this by not exploring some redundant interleavings, more accurately, by shrinking the set of successors of a state while preserving system behavior. Existing algorithms are designed for (unbounded) asynchronous systems and do not directly adapt to our kind of models (see below). Although, at the moment, we are unable to suggest POR techniques for bounded asynchrony, we propose an algorithm that exploits the restricted communication encountered in systems that model cell-cell interaction, we refer to our algorithm as *communication based reduction*, or CBR for short. Like POR, our algorithm searches only some of the possible interleavings. For every interleaving, our algorithm explores an interleaving that visits the same states on a macro-step level. We reduce the reachable region of the scheduler from exponential size to polynomial size in the number of processes, and thus we have a direct and important impact on enumerative model checking. Our approach is applicable to all linear time properties whose validity is preserved by restricting attention to macro steps. Much like POR, the *next* operator cannot be handled. In particular, every property that relates to a single process (without next), and Boolean combinations of such properties, retain their validity.

3.1 Communication Based Reduction

The explicit scheduler S defined in Subsection 2.2 allows all possible interleavings of processes within a macro-step. We prove that we can construct a new scheduler that preserves system macro behavior (macro-step level behavior) but allows fewer interleavings. Let $\mathcal{P} = P_1^1 \|_{ba} \cdots \|_{ba} P_n^1$ be the bounded asynchronous composition of $P_1, P_2, \ldots, P_n$ according to rate 1 (see Section 2).[7]
We first formally define a *macro-step* g of $\mathcal{P}$ as a sequence of states $g : s = s_0, s_1, \ldots, s_m$ satisfying:

- g is a subsequence of a run,
- s_0 is initial with respect to the scheduler, i.e., $\overline{s_0[b_k]}$ holds for all $0 \leq k \leq n$,
- s_m is final with respect to the scheduler, i.e., $s_m[b_k]$ holds for all $0 \leq k \leq n$,
- s_m is the only final state in g.

A macro-step induces a total and a partial order over the processes of $\mathcal{P}$. The total order represents the order in which the processes move and we refer to it as the macro-step's interleaving. The partial order represents the order in which processes pass messages (via variables) and we refer to it as the macro-steps channel configuration.

Consider a macro-step $g : s = s_0, s_1, \ldots, s_m$ of $\mathcal{P}$. The *interleaving* of g, denoted $\mathcal{I}_g = (<_{Ig}, =_{Ig})$, is an order such that: $(P_k <_{Ig} P_l)$ if there exists s_i in g such that $s_i[b_k]\overline{s_i[b_l]}$ and $(P_k =_{Ig} P_l)$ if $(P_k \not<_{Ig} P_l) \wedge (P_l \not<_{Ig} P_k)$. That is, $(P_k <_{Ig} P_l)$ if P_k moves before P_l in the interleaving g.

We say that there is a communication channel c_{kl} connecting P_k and P_l if $V_k \cap V_l \neq \emptyset$. The *neighbor order* of g, denoted $(<_{Ng}, =_{Ng})$, is the partial order defined as the restriction of the interleaving of g to the neighboring processes. $P_k <_{Ng} P_l$ iff

[7] Here, we only describe the case of processes running at equal rates. The same ideas can be easily extended to general rates.

$P_k <_{Ig} P_l$ and there exists a channel c_{kl}. We define in a similar way $=_{Ng}$. The *channel configuration* of g, denoted $(<_{Cg}, =_{Cg})$, is the transitive closure of the neighbor order. That is, $P_k <_{Cg} P_l$ if a change in value of a variable of P_k in interleaving g can be sensed by P_l in the same interleaving.

Given a macro-step g, a channel c_{kl} may have one of three states: enabled from k to l, if $P_k <_{Cg} P_l$, enabled from l to k, if $P_l <_{Cg} P_k$, disabled, if $P_l =_{Cg} P_l$. Intuitively, a channel is enabled if it may propagate a value generated in the current macro-step.

Two interleavings are *$\mathcal{P}$-equivalent* if they induce the same channel configuration.

Within $\mathcal{P}$, we say that t is a *macro-successor* of s with respect to interleaving $\mathcal{I}$ if there exists a macro-step g with initial state s, interleaving $\mathcal{I}$ and final state t.

The following lemma establishes that two equivalent interleavings have the same set of macro-successors.

Lemma 1. *Consider two $\mathcal{P}$-equivalent interleavings $\mathcal{I}$ and $\mathcal{I}'$. If s' is a macro-successor of s with respect to $\mathcal{I}$, then s' is a macro-successor of s with respect to $\mathcal{I}'$.*

A scheduler that allows only one of two $\mathcal{P}$-equivalent interleavings preserves system macro-behavior. It follows that a scheduler that generates only one interleaving per channel configuration produces a correct macro-state behavior.

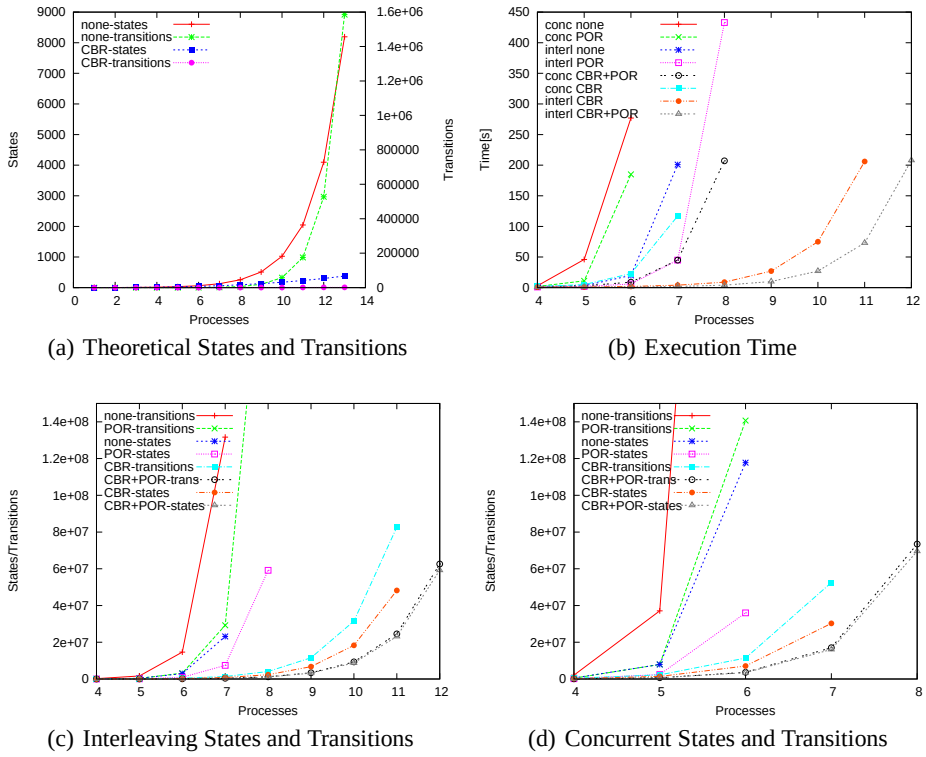
Here after we focus on the case of *line communication scheme* ($V_k \cap V_l = \emptyset$, for all $l \notin \{k-1, k+1\}$, $k \in (1..n)$). This is a common configuration in biological models where communication is very local. Extension to 2-dimensional configurations follows similar ideas.

Let c_k denote the channel $c_{k,k+1}$. In interleaving g, channel c_k is *enabled-right* if enabled from k to $k+1$, *enabled-left* if enabled from $k+1$ to k, and disabled as before.

Given a channel configuration we construct one interleaving that preserves it. Let $c_{r_0}, c_{r_1}, \ldots, c_{r_{mr}}$ be the right-enabled channels. Process P_{r_0} is oblivious to whatever happens in the same macro step in processes $P_{r_0+1}, \ldots, P_n$ because its communication with these processes happens through process P_{r_0+1} which moves after it. Thus, whatever actions are performed by processes $P_{r_0+1}, \ldots, P_n$ they do not affect the actions of processes $P_1, \ldots, P_{r_0}$. We may shuffle all the actions of processes $P_1, \ldots, P_{r_0}$ to the beginning of the interleaving preserving the right-enabled channel c_{r_0}. The new interleaving starts by handling all processes $P_1, \ldots, P_{r_0}$ from right to left. Let $c_{l_0}, c_{l_1}, \ldots, c_{l_{ml}}$ be the left-enabled channels in $1, \ldots, r_0$. Then, the order of moves is: first processes $P_{l_m+1}, \ldots, P_{r_0}$, then $P_{l_{m-1}+1}, \ldots, P_{l_m}$, and so on until $P_0, \ldots P_{l_1}$.

Next, using the same reasoning, we can handle the processes in the range $r_0 + 1, \ldots, r_1$ from right to left according to the left-enabled channels, and so on.

The CBR scheduler also uses the Boolean variables $b_1, \ldots, b_n$, however, the possible assignments are those where the processes can be partitioned to at most four maximal groups of consecutive processes that have either moved or not. More formally, we denote the value of $b_1, \ldots, b_n$ by a sequence of 0 and 1, then the configurations can be described by the following regular expressions: $0^+1^+0^+$, 1^+0^+, $1^+0^+1^+0^+$, and 1^+. With similar intuition configurations of the form 0^+, 0^+1^+, and $1^+0^+1^+$ are also reachable. For example, in a system with 6 processes the configurations 000111 and 110110 are reachable while the configuration 010101 is not. There are only $O(n^3)$ such reachable states, compared to 2^n reachable states in the original scheduler. Fig. 2(a)



(a) Theoretical States and Transitions (b) Execution Time

(c) Interleaving States and Transitions (d) Concurrent States and Transitions

Fig. 2. Comparing theory and practice

compares the number of states and transitions of the two schedulers (none, the scheduler described in Section 2 with no reduction vs. CBR, the scheduler described above) for different number of processes.

3.2 Experimental Evaluation

We compare experimentally the performance of the CBR scheduler with POR methods. We translate the model in Fig. 1 to Promela and use Spin [17] for a thorough analysis of the behavior of CBR.

We explain, intuitively, why POR is inappropriate for bounded asynchrony. We assume basic familiarity with POR. First, we find it very important that processes may move concurrently under bounded asynchrony. POR is developed for 'classical' asynchronous systems, thus, it does not allow for processes to move concurrently. Second, a macro-step in bounded asynchrony is a sequence of at most n local steps, and noticing that one interleaving is redundant may require exploration of more than 1 lookahead. Let us further explore this with an example. Suppose that we give up on concurrent moves and would like to use POR for reasoning about the same bounded asynchronous system. That is, processes in a line configuration where only neighbor processes may

communicate. In the beginning of a macro-step, all processes are enabled. Communication between processes implies that we cannot find independent processes (such that the order of scheduling them does not matter), and we have to explore all possible n processes as the first process to move. With one process ahead of others, it is clear that the processes to the left of this process and to its right are no longer connected and the order between scheduling every process on the left and every process on the right can be exchanged. However, among the processes on one side, there is still dependency and the same selection by the scheduler has to be applied recursively. Overall, the number of possible interleavings to be checked is still exponential in n.[8] As exhibited by our experiments, POR does offer some reduction, however, this cannot be compared to the order of magnitude saving offered by using communication-based reduction.

We consider the bounded asynchronous composition of n cells in a line configuration. All processes start from the same state. If we disallow concurrent moves, we verify that there are no adjacent cells that assume fate1 (see Fig. 1). We add a mechanism that allows us to model concurrent moves using Spin's interleaving semantics. This mechanism consists of deciding to store the next values of variables in a local copy, allowing other processes to perform a computation according to the old values, and finally updating the new values. Obviously, this mechanism increases considerably the number of states in the system. For this case we verify that a cell assumes fate2 only if it has a neighbor that assumes fate1. We evaluate the CBR scheduler by considering the time for enumerative model checking and the number of states and transitions explored during model checking. We compare the behavior of the CBR scheduler with the basic scheduler described in Section 2 (simple scheduler) when POR is enabled and disabled. We perform two sets of experiments, both using Spin. The first set of experiments uses the normal interleaving semantics of Spin. In this case the size of the CBR scheduler is reduced from $O(n^3)$ to $O(n^2)$ states. This set of experiments includes running the simple scheduler without any reductions (none), the simple scheduler with POR (POR), the CBR scheduler (CBR), and the CBR scheduler with POR (CBR+POR). The second set of experiments includes a mechanism that makes Spin mimic the possibility of concurrent moves. We note that this additional mechanism increases the size of each process and that in order to communicate with the CBR scheduler each process has additional variables. Thus, the experiment is unfair with respect to the CBR scheduler. As before, this set of experiments includes running the simple scheduler (conc none), simple scheduler with POR (conc POR), CBR scheduler (conc CBR), and CBR scheduler with POR (conc CBR+POR). In all experiments, increasing the number of processes by one leads to memory overflow (10GB). For example, for the experiment with 9 processes, with the simple scheduler where POR is enabled, Spin requires more than 10GB of memory. Fig. 2(b) compares the model-checking time for the different experiments. Figures 2(c) and 2(d) compare the numbers of states and transitions explored in the first (interleaving semantics) and second (with mechanism mimicking concurrent moves) sets of experiments, respectively. For better scaling, the range of values covered by these figures does not include the number of transitions for the none-experiments in the cases of 7 and 5 processes, respectively. Notice that the size of the system it-

[8] More accurately, the analysis is as follows. The number of interleavings of one process is $f(1) = 1$, the number of interleavings of zero processes is $f(0) = 0$. Generally, $f(n) = \Sigma_{i=1}^{n}(f(i-1) + f(n-i)) = 2f(n-1) + f(n-1) = 3f(n-1)$ and $f(n) = 2 \cdot 3^{n-2}$.

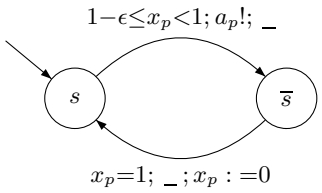


Fig. 3. CTS for one rate

self increases exponentially with the number of processes. The experiments confirm that POR offers some improvement while the communication-based reduction affords a significant improvement when compared with the simple scheduler with POR.

The success of CBR in the context of bounded asynchrony suggests that it may be useful to analyze the communication structure in systems prior to model checking and to apply specific optimizations based on this analysis. Further research in this direction is out of the scope of this paper.

4 A Possible Mechanistic Explanation for Bounded Asynchrony

It is rather obvious that a scheduler such as the one we describe in Section 2 does not exist in real biological systems. While trying to describe biological behavior (of this type) in high-level requires us to use a notion like bounded asynchrony, it is not clear what is responsible for this kind of behavior in real systems. Obviously, no centralized control exists in this case, and there has to be some distributed mechanism that creates this kind of behavior. In this section we show that bounded asynchrony can be naturally used to abstract a special kind of distributed real-time mechanism. Thus, in some cases, similar scheduling mechanisms can be used to construct rough abstractions of real-time systems. From a biological point of view, it is an interesting challenge to design biological experiments that will confirm or falsify the hypothesis that internal clock-like mechanisms are responsible for the emerging behavior of bounded asynchrony.

We suggest clocked transition systems (CTS) as a possible distributed mechanism that produces bounded asynchrony. The systems we consider use a single clock, perform actions when this clock reaches a certain value, and reset the clock. We give a high-level description of the CTS we have in mind.

Consider the CTS Φ depicted in Fig. 3. The CTS has two Boolean variables s and a_p and one clock x_p. The values of s correspond to the two states in the figure. The CTS is allowed to move from s to $\overline{s}$ when the clock x is in the range $[1-\epsilon, 1)$, for some ϵ. When the CTS moves from $\overline{s}$ to s, it resets the clock back to 0. The variable a_p is the scheduling variable that this CTS sets; it changes when the system moves from s to $\overline{s}$, and does not change when the system moves from $\overline{s}$ to s. The possible computations of this system include the clock progressing until some point in $[1-\epsilon, 1)$, then the system makes a transition from s to $\overline{s}$ while changing a_p, then the clock progresses until it is 1, and finally the system makes a transition from $\overline{s}$ to s. Then, the process repeats itself when the global time is $[2-\epsilon, 2)$, $[3-\epsilon, 3)$, and in general $[i-\epsilon, i)$ for every i.

Consider now the composition of Φ with a TS P that uses a_p as its scheduling variable. The composition of the two is a CTS in which moves of the TS P happen in

the time range $[i-\epsilon, i)$ for every $i \in \mathbb{N}$. Suppose that we have two TS P and Q with scheduling variables a_p and a_q, respectively. We take the composition of two CTS as above using clocks x_p and x_q and the variables a_p and a_q. It follows that P and Q take approximately one time unit to make one move. However, the exact timing is not set. In a run of the system combined of the four CTSs the order of actions between P and Q is not determined. Every possible ordering of the actions is possible. In addition, the transitions that reset the clocks x_p and x_q ensure that the two TSs stay coupled. However long the execution, it cannot be the case that P takes significantly more actions than Q (in this case more than one). Under appropriate projection, the sequence of actions taken by the composition of the four systems, is equivalent to the sequence of actions taken by the bounded-asynchronous composition of P and Q with rate 1.

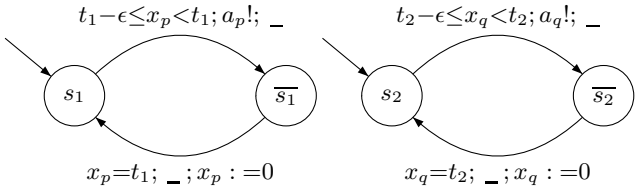


Fig. 4. CTSs for different rates

We now turn to consider the more general scheduler. Consider the CTSs in Fig. 4. They resemble the simple CTS presented above, however use the bounds of t_1 and t_2 time units, respectively. Denote the CTS using bound t_1 by Φ_1, and the CTS using bound t_2 by Φ_2. A computation of Φ_1 is a sequence of steps where time progresses until the range $[i \cdot t_1 - \epsilon, i \cdot t_1)$, then the system takes a step, then the time progresses until $i \cdot t_1$, and the system takes a step that resets the local clock. A computation of Φ_2 is similar, with t_2 replacing t_1.

Let P and Q be two TSs with scheduling variables a_p and a_q as above. Consider the composition of P and Q with Φ_1 and Φ_2. It follows that P moves every t_1 time units and Q every t_2 time units[9]. Every t_1 time units P performs an action, and every t_2 time units Q performs an action. At time t such that both t_1 and t_2 divide t, both P and Q make moves, however, the order between P and Q is not determined. We can show that under appropriate projection, the sequence of actions taken by the composition of the four systems, is equivalent to the sequence of actions taken by the bounded-asynchronous composition of P and Q with rates t_1 and t_2, respectively.

We note that the CTSs have their resets set at exact time points, suggesting that a composition of such systems requires a central clock. We can still maintain 'bounded-asynchronous' behavior if the reset occurs concurrently with the system, however, maintaining ϵ small enough and restricting the number of steps made by the system. For example, if ϵ is $1/100$, then regardless of the exact behavior, the first 98 macro-steps still

[9] For every ϵ and for every values t_1 and t_2 there are some integers i_1 and i_2 such that $[i_1 \cdot t_1 - \epsilon, i_1 \cdot t_1]$ intersects $[i_2 \cdot t_2 - \epsilon, i_2 \cdot t_2]$. As we are interested only in the sequence of actions taken by p_1 and p_2, restricting t_1 and t_2 to range over integer values seems reasonable.

respect bounded asynchrony. It follows, that unsynchronized local clocks augmented by frequent enough synchronizations would lead to the exact same behavior. It is an interesting question whether similar ideas can be used for the abstraction of real time and probabilistic systems.

5 Acknowledgments

We thank Alex Hajnal for fruitful discussions, and Marc Schaub for comments on an earlier draft of the manuscript.

References

1. http://mtc.epfl.ch/∼piterman/bio.
2. R. Alur, C. Belta, F. Ivancic, V. Kumar, M. Mintz, G. Pappas, H. Rubin, and J. Schug. Hybrid modeling and simulation of biomolecular networks. In *Fourth International Workshop on Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 19–32. Springer-Verlag, 2001.
3. J. Barjis and I. Barjis. Formalization of the protein production by means of petri nets. In *Proceedings International Conference on Information Intelligence Systems*, pages 4–9. IEEE, 1999.
4. G. Bernot, J.P. Comet, A. Richard, and J. Guespin. Application of formal methods to biological regulatory networks: extending thomas asynchronous logical approach with temporal logic. *Journal of Theoretical Biology*, 229(3):339–347, 2004.
5. L. Calzone, N. Chabrier-Rivier, F. Fages, and S. Soliman. Machine learning biochemical networks from temporal logic properties. *Transactions on Computational Systems Biology*, 4:68–94, 2006.
6. E.C. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
7. D. Dill, M.A. Knapp, P. Gage, C. Talcott, K. Laderoute, and P. Lincoln. The pathalyzer: a tool for analysis of signal transduction pathways. In *Proceedings of the First Annual Recomb Satellite Workshop on Systems Biology*, 2005.
8. S. Efroni, D. Harel, and I. R. Cohen. Toward rigorous comprehension of biological complexity: modeling, execution, and visualization of thymic T-cell maturation. *Genome Res*, 13(11):2485–97, 2003.
9. S. Efroni, D. Harel, and I.R. Cohen. Emergent dynamics of thymocyte development and lineage determination. *PLoS Computational Biology*, 3(1):127–136, 2007.
10. J. Fisher and T.A. Henzinger. Executable cell biology. *Nature Biotechnology*, 25(11):1239–49, 2007.
11. J. Fisher, N. Piterman, A. Hajnal, and T.A. Henzinger. Predictive modeling of signalling crosstalk during *C. elegans* vulval development. *PLoS Computational Biology*, 3(5):e92, 2007.
12. J. Fisher, N. Piterman, E. J. Hubbard, M. J. Stern, and D. Harel. Computational insights into *Caenorhabditis elegans* vulval development. *Proc Natl Acad Sci U S A*, 102(6):1951–6, 2005.
13. R. Ghosh and C. Tomlin. Lateral inhibition through delta-notch signaling: A piecewise affine hybrid model. In *4th International Workshop on Hybrid Systems Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 232–246, Rome, Italy, 2001.
14. D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. of Phys. Chemistry*, 81(25):2340–61, 1977.
15. J. Heath, M. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn. Probabilistic model checking of complex biological pathways. *Theoretical Computer Science*, 2008. Special issue on Converging Sciences: Informatics and Biology. To appear.

16. T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proc. 1st International Workshop on Embedded Software*, volume 2211 of *Lect. Notes in Comp. Sci.*, pages 166–184. Springer-Verlag, 2001.
17. G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
18. N. Kam, D. Harel, and I.R. Cohen. The immune system as a reactive system: Modeling T-cell activation with statecharts. In *IEEE Symposium of Visual Languages and Formal Methods*, pages 15–22., 2003.
19. N. Kam, D. Harel, H. Kugler, R. Marelly, A. Pnueli, E. J. A. Hubbard, and M. J. Stern. Formal modeling of C. elegans development: A scenario-based approach. In *First International Workshop on Computational Methods in Systems Biology*, volume 2602 of *Lecture Notes in Computer Science*, pages 4–20, Roverto, Italy, 2003. Springer-Verlag.
20. S. A. Kauffman. Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology*, 22(3):437–467, 1969.
21. Y. Kesten and A. Pnueli. Control and data abstractions: The cornerstones of practical formal verification. *Software Tools for Technology Transfer*, 2(1):328–342, 2000.
22. C. Priami, A. Regev, E.Y. Shapiro, and W Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80(1):25–31, 2001.
23. A. Sadot, J. Fisher, D. Barak, Y. Admanit, M. J. Stern, E. J. Hubbard, and D. Harel. Towards verified biological models. *IEEE Transactions in Computational Biology and Bioinformatics*, 2007. To appear.