The Combinatorics of Barrier Synchronization*

Olivier Bodini[1], Matthieu Dien[2], Antoine Genitrini[3], and Frédéric Peschanski[3]

[1] Université Paris-Nord – LIPN – CNRS UMR 7030
`Olivier.Bodini@lipn.univ-paris13.fr`
[2] Université de Caen – GREYC – CNRS UMR 6072
`Matthieu.Dien@unicaen.fr`
[3] Sorbonne University – LIP6 – CNRS UMR 7607
`{Antoine.Genitrini,Frederic.Peschanski}@lip6.fr`

Abstract. In this paper we study the notion of synchronization from the point of view of combinatorics. As a first step, we address the quantitative problem of counting the number of executions of simple processes interacting with synchronization barriers. We elaborate a systematic decomposition of processes that produces a symbolic integral formula to solve the problem. Based on this procedure, we develop a generic algorithm to generate process executions uniformly at random. For some interesting sub-classes of processes we propose very efficient counting and random sampling algorithms. All these algorithms have one important characteristic in common: they work on the control graph of processes and thus do not require the explicit construction of the state-space.

Keywords: Barrier synchronization · Combinatorics · Uniform random generation.

1 Introduction

The objective of our (rather long-term) research project is to study the *combinatorics* of concurrent processes. Because the mathematical toolbox of combinatorics imposes strong constraints on what can be modeled, we study *process calculi* with a very restricted focus. For example in [5] the processes we study can only perform atomic actions and fork child processes, and in [4] we enrich this primitive language with *non-determinism*. In the present paper, our objective is to isolate another fundamental "feature" of concurrent processes: *synchronization*. For this, we introduce a simple process calculus whose only non-trivial concurrency feature is a principle of *barrier synchronization*. This is here understood intuitively as the single point of control where multiple processes have to "meet" before continuing. This is one of the important building blocks for concurrent and parallel systems [13].

Combinatorics is about "counting things", and what we propose to count in our study is the number of executions of processes wrt. their "syntactic size".

* This research was partially supported by the ANR MetACOnc project ANR-15-CE40-0014.

This is a symptom of the so-called "combinatorial explosion", a defining characteristic of concurrency. As a first step, we show that counting executions of concurrent processes is a difficult problem, even in the case of our calculus with limited expressivity. Thus, one important goal of our study is to investigate interesting sub-classes for which the problem becomes "less difficult". To that end, we elaborate in this paper a systematic decomposition of arbitrary processes, based on only four rules: (B)ottom, (I)ntermediate, (T)op and (S)plit. Each rule explains how to "remove" one node from the control graph of a process while taking into account its contribution in the number of possible executions. Indeed, one main feature of this BITS-decomposition is that it produces a symbolic integral formula to solve the counting problem. Based on this procedure, we develop a generic algorithm to generate process executions uniformly at random. Since the algorithm is working on the control graph of processes, it provides a way to statistically analyze processes without constructing their state-space explicitly. In the worst case, the algorithm cannot of course overcome the hardness of the problem it solves. However, depending on the rules allowed during the decomposition, and also on the strategy adopted, one can isolate interesting sub-classes wrt. the counting and random sampling problem. We identify well-known "structural" sub-classes such as *fork-join parallelism* [11] and *asynchronous processes with promises* [15]. For some of these sub-classes we develop dedicated and efficient counting and random sampling algorithms. A large sub-class that we find particularly interesting is what we call the "BIT-decomposable" processes, i.e. only allowing the three rules (B), (I) and (T) in the decomposition. The counting formula we obtain for such processes is of a linear size (in the number of atomic actions in the processes, or equivalently in the number of vertices in their control graph). We also discuss informally the typical shape of "BIT-free" processes.

The outline of the paper is as follows. In Section 2 we introduce a minimalist calculus of barrier synchronization. We show that the control graphs of processes expressed in this language are isomorphic to arbitrary partially ordered sets (Posets) of atomic actions. From this we deduce our rather "negative" starting point: counting executions in this simple language is intractable in the general case. In Section 3 we define the BITS-decomposition, and we use it in Section 4 to design a generic uniform random sampler. In Section 5 we discuss various sub-classes of processes related to the proposed decomposition, and for some of them we explain how the counting and random sampling problem can be solved efficiently. In Section 6 we propose an experimental study of the algorithm toolbox discussed in the paper.

Note that some technical complement and proof details are deferred to an external "companion" document. Moreover we provide the full source code developed in the realm of this work, as well as the benchmark scripts. All these complement informations are available online[4].

[4] cf. https://gitlab.com/ParComb/combinatorics-barrier-synchro.git

Related work

Our study intermixes viewpoints from concurrency theory, order-theory as well as combinatorics (especially enumerative combinatorics and random sampling). The *heaps combinatorics* (studied in e.g. [1]) provides a complementary interpretation of concurrent systems. One major difference is that this concerns "true concurrent" processes based on the trace monoid, while we rely on the alternative *interleaving semantics*. A related uniform random sampler for networks of automata is presented in [3]. Synchronization is interpreted on words using a notion of "shared letters". This is very different from the "structural" interpretation as joins in the control graph of processes. For the generation procedure [1] requires the construction of a "product automaton", whose size grows exponentially in the number of "parallel" automata. By comparison, all the algorithms we develop are based on the control graph, i.e. the space requirement remains polynomial (unlike, of course, the time complexity in some cases). Thus, we can interpret this as a space-time trade-of between the two approaches. A related approach is that of investigating the combinatorics of *lassos*, which is connected to the observation of state spaces through linear temporal properties. A uniform random sampler for lassos is proposed in [16]. The generation procedure takes place *within* the constructed state-space, whereas the techniques we develop do not require this explicit construction. However lassos represent infinite runs whereas for now we only handle finite (or finite prefixes) of executions.

A coupling from the past (CFTP) procedure for the uniform random generation of linear extensions is described, with relatively sparse details, in [14]. The approach we propose, based on the continuous embedding of Posets into the hypercube, is quite complementary. A similar idea is used in [2] for the enumeration of Young tableaux using what is there called the *density method*. The paper [12] advocates the uniform random generation of executions as an important building block for *statistical model-checking*. A similar discussion is proposed in [18] for *random testing*. The *leitmotiv* in both cases is that generating execution paths *without* any bias is difficult. Hence a uniform random sampler is very likely to produce interesting and complementary tests, if comparing to other test generation strategies.

Our work can also be seen as a continuation of the *algorithm and order* studies [17] orchestrated by Ivan Rival in late 1980's only with powerful new tools available in the modern combinatorics toolbox.

2 Barrier synchronization processes

The starting point of our study is the small process calculus described below.

Definition 1 (Syntax of barrier synchronization processes). *We consider countably infinite sets $\mathcal{A}$ of (abstract) atomic actions, and $\mathcal{B}$ of barrier names. The set $\mathcal{P}$ of processes is defined by the following grammar:*

$$\begin{array}{rlll} P,Q & ::= & 0 & \textit{(termination)} \\ & | & \alpha.P & \textit{(atomic action and prefixing)} \\ & | & \langle B\rangle P & \textit{(synchronization)} \\ & | & \nu(B)P & \textit{(barrier and scope)} \\ & | & P \parallel Q & \textit{(parallel)} \end{array}$$

The language has very few constructors and is purposely of limited expressivity. Processes in this language can only perform atomic actions, fork child processes and interact using a basic principle of *synchronization barrier*. A very basic process is the following one:

$$\nu(B)\ \ [\mathsf{a}_1.\langle B\rangle\ \mathsf{a}_2.0 \parallel \langle B\rangle \mathsf{b}_1.0 \parallel \mathsf{c}_1.\langle B\rangle\ 0]$$

This process can initially perform the actions a_1 and c_1 in an arbitrary order. We then reach the state in which all the processes agrees to synchronize on B:

$$\nu(B)\ \ [\langle B\rangle\ \mathsf{a}_2.0 \parallel \langle B\rangle \mathsf{b}_1.0 \parallel \langle B\rangle\ 0]$$

The possible next transitions are: $\xrightarrow{\mathsf{a}_2} \mathsf{b}_1.0 \xrightarrow{\mathsf{b}_1} 0$, alternatively $\xrightarrow{\mathsf{b}_1} \mathsf{a}_2.0 \xrightarrow{\mathsf{a}_2} 0$ In the resulting states, the barrier B has been "consumed".

The operational semantics below characterize processes transitions of the form $P \xrightarrow{\alpha} P'$ in which P can perform action α to reach its (direct) derivative P'.

Definition 2 (Operational semantics).

$$\frac{}{\alpha.P \xrightarrow{\alpha} P}\ \textit{(act)} \qquad \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q}\ \textit{(lpar)} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \parallel Q \xrightarrow{\alpha} P \parallel Q'}\ \textit{(rpar)}$$

$$\frac{\mathsf{sync}_B(P){=}Q \quad \mathsf{wait}_B(Q) \quad P \xrightarrow{\alpha} P'}{\nu(B)P \xrightarrow{\alpha} \nu(B)P'}\ \textit{(lift)} \qquad \frac{\mathsf{sync}_B(P){=}Q \quad \neg\mathsf{wait}_B(Q) \quad Q \xrightarrow{\alpha} Q'}{\nu(B)P \xrightarrow{\alpha} Q'}\ \textit{(sync)}$$

$$\textit{with:}\ \left[\begin{array}{l} \mathsf{sync}_B(0){=}0 \\ \mathsf{sync}_B(\alpha.P){=}\alpha.P \\ \mathsf{sync}_B(P\|Q){=}\mathsf{sync}_B(P)\|\mathsf{sync}_B(Q) \\ \mathsf{sync}_B(\nu(B)P){=}\nu(B)P \\ \forall C{\neq}B,\ \mathsf{sync}_B(\nu(C)P){=}\nu(C)\ \mathsf{sync}_B(P) \\ \mathsf{sync}_B(\langle B\rangle P){=}P \\ \forall C{\neq}B,\ \mathsf{sync}_B(\langle C\rangle P){=}\langle C\rangle P \end{array}\right. \quad \left[\begin{array}{l} \mathsf{wait}_B(0){=}\mathit{false} \\ \mathsf{wait}_B(\alpha.P){=}\mathsf{wait}_B(P) \\ \mathsf{wait}_B(P\|Q){=}\mathsf{wait}_B(P){\vee}\mathsf{wait}_B(Q) \\ \mathsf{wait}_B(\nu(B)P){=}\mathit{false} \\ \forall C{\neq}B,\ \mathsf{wait}_B(\nu(C)P){=}\mathsf{wait}_B(P) \\ \mathsf{wait}_B(\langle B\rangle P){=}\mathit{true} \\ \forall C{\neq}B,\ \mathsf{wait}_B(\langle C\rangle P){=}\mathsf{wait}_B(P) \end{array}\right.$$

The rule (sync) above explains the synchronization semantics for a given barrier B. The rule is non-trivial given the broadcast semantics of barrier synchronization. The definition is based on two auxiliary functions. First, the function $\mathsf{sync}_B(P)$ produces a derivative process Q in which all the possible synchronizations on barrier B in P have been effected. If Q has a sub-process that cannot yet synchronize on B, then the predicate $\mathsf{wait}_B(Q)$ is true and the synchronization on B is said incomplete. In this case the rule (sync) does not apply, however the transitions *within* P can still happen through (lift).

2.1 The control graph of a process

We now define the notion of a (finite) execution of a process.

Definition 3 (execution). *An* execution σ *of* P *is a finite sequence* $\langle \alpha_1, \ldots, \alpha_n \rangle$ *such that there exist a set of processes* $P'_{\alpha_1}, \ldots, P'_{\alpha_n}$ *and a path* $P \xrightarrow{\alpha_1} P'_{\alpha_1} \ldots \xrightarrow{\alpha_n} P'_{\alpha_n}$ *with* $P'_{\alpha_n} \nrightarrow$ *(no transition is possible from* P'_{α_n}*).*

We assume that the occurrences of the atomic actions in a process expression have all distinct labels, $\alpha_1, \ldots, \alpha_n$. This is allowed since the actions are uninterpreted in the semantics (cf. Definition 2). Thus, each action α in an execution σ can be associated to a unique *position*, which we denote by $\sigma(\alpha)$. For example if $\sigma = \langle \alpha_1, \ldots, \alpha_k, \ldots, \alpha_n \rangle$, then $\sigma(\alpha_k) = k$.

The behavior of a process can be abstracted by considering the *causal ordering relation* wrt. its atomic actions.

Definition 4 (cause, direct cause). *Let* P *be a process. An action* α *of* P *is said a* cause *of another action* β*, denoted by* $\alpha < \beta$*, iff for any execution* σ *of* P *we have* $\sigma(\alpha) < \sigma(\beta)$*. Moreover,* α *is a* direct cause *of* β*, denoted by* $\alpha \prec \beta$ *iff* $\alpha < \beta$ *and there is no* γ *such that* $\alpha < \gamma < \beta$*. The relation* $<$ *obtained from* P *is denoted by* $\mathscr{PO}(P)$*.*

$$\nu(B) \begin{bmatrix} \alpha_1.\langle B \rangle \parallel \alpha_2.\langle B \rangle \parallel \ldots \parallel \alpha_n.\langle B \rangle \\ \langle B \rangle.\beta_1 \parallel \langle B \rangle.\beta_2 \parallel \ldots \parallel \langle B \rangle.\beta_n \end{bmatrix}$$

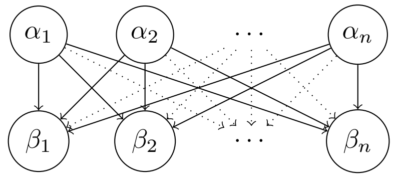


Fig. 1. A process of size $2n$ and its control graph with $2n$ nodes and n^2 edges.

Obviously $\mathscr{PO}(P)$ is a *partially ordered set* (poset) with *covering* $\prec$, capturing the *causal ordering* of the actions of P. The covering of a partial order is by construction an *intransitive directed acyclic graph* (DAG), hence the description of $\mathscr{PO}(P)$ itself is simply the transitive closure of the covering, yielding $O(n^2)$ edges over n elements. The worst case (maximizing the number of edges) is a complete bipartite graph with two sets of $2n$ vertices connected by n^2 edges (cf. Fig. 1).
For most practical concerns we will only consider the covering, i.e. the intransitive DAG obtained by the *transitive reduction* of the order. It is possible to direclty construct this control graph, according to the following definition.

Definition 5 (Construction of control graphs). *Let* P *be a process term. Its control graph is* $\mathsf{ctg}(P) = \langle V, E \rangle$*, constructed inductively as follows:*

$$\left[\begin{array}{l|l} \mathsf{ctg}(0) = \langle \emptyset, \emptyset \rangle & \mathsf{ctg}(\nu(B)P) = \bigotimes_{\langle B \rangle} \mathsf{ctg}(P) \\ \mathsf{ctg}(\alpha.P) = \alpha \rightsquigarrow \mathsf{ctg}(P) & \mathsf{ctg}(P \parallel Q) = \mathsf{ctg}(P) \cup \mathsf{ctg}(Q) \\ \mathsf{ctg}(\langle B \rangle P) = \langle B \rangle \rightsquigarrow \mathsf{ctg}(P) & \quad \textit{with } \langle V_1, E_1 \rangle \cup \langle V_2, E_2 \rangle = \langle V_1 \cup V_2, E_1 \cup E_2 \rangle \end{array}\right.$$

$$\textit{with} \begin{cases} x \rightsquigarrow \langle V, E \rangle = \langle V \cup \{x\}, \{(x, y) \mid y \in \mathsf{srcs}(E) \vee (E = \emptyset \wedge y \in V)\} \rangle \\ \mathsf{srcs}(E) = \{y \mid (y, z) \in E \wedge \nexists x,\ (x, y) \in E\} \\ \bigotimes_{\langle B \rangle} \langle V, E \rangle = \langle V \setminus \{\langle B \rangle\}, E \setminus \{(x, y) \mid x \neq y \wedge (x = \langle B \rangle \vee y = \langle B \rangle)\} \\ \qquad\qquad\qquad\qquad \cup \{(\alpha, \beta) \mid \{(\alpha, \langle B \rangle),\ (\langle B \rangle, \beta)\} \subseteq E\} \rangle \end{cases}$$

Given a control graph Γ, the notation $x \rightsquigarrow \Gamma$ corresponds to prefixing the graph by a single atomic action. The set $\mathsf{srcs}(E)$ corresponds to the *sources* of the edges in E, i.e. the vertices without an incoming edge. And $\bigotimes_{\langle B\rangle} \Gamma$ removes an explicit barrier node and connect all the processes ending in B to the processes starting from it. In effect, this realizes the synchronization described by the barrier B. We illustrate the construction on a simple process below:

$$\begin{aligned}
&\mathsf{ctg}(\nu(B)\nu(C)[\langle B\rangle\langle C\rangle a.0 \| \langle B\rangle\langle C\rangle b.0]) \\
&= \textstyle\bigotimes_{\langle B\rangle}\bigotimes_{\langle C\rangle} (\mathsf{ctg}(\langle B\rangle\langle C\rangle a.0) \cup \mathsf{ctg}(\langle B\rangle\langle C\rangle b.0)) \\
&= \textstyle\bigotimes_{\langle B\rangle}\bigotimes_{\langle C\rangle} \langle\{\langle B\rangle, \langle C\rangle, a\}, \{(\langle B\rangle, \langle C\rangle), (\langle C\rangle, a)\}\rangle \\
&\qquad \cup \langle\{\langle B\rangle, \langle C\rangle, b\}, \{(\langle B\rangle, \langle C\rangle), (\langle C\rangle, b)\}\rangle \\
&= \textstyle\bigotimes_{\langle B\rangle}\bigotimes_{\langle C\rangle} \langle\{\langle B\rangle, \langle C\rangle, a, b\}, \{(\langle B\rangle, \langle C\rangle), (\langle C\rangle, a), (\langle C\rangle, b)\}\rangle \\
&= \textstyle\bigotimes_{\langle B\rangle} \langle\{\langle B\rangle, a, b\}, \{(\langle B\rangle, a), (\langle B\rangle, b)\}\rangle \\
&= \langle\{a, b\}, \emptyset\rangle
\end{aligned}$$

The graph with only two unrelated vertices and no edge is the correct construction. Now, slightly changing the process we see how the construction fails for deadlocked processes.

$$\begin{aligned}
&\mathsf{ctg}(P) = \textstyle\bigotimes_{\langle B\rangle}\bigotimes_{\langle C\rangle} (\mathsf{ctg}(\langle B\rangle\langle C\rangle a.0) \cup \mathsf{ctg}(\langle C\rangle\langle B\rangle b.0)) \\
&= \textstyle\bigotimes_{\langle B\rangle}\bigotimes_{\langle C\rangle} \langle\{\langle B\rangle, \langle C\rangle, a\}, \{(\langle B\rangle, \langle C\rangle), (\langle C\rangle, a)\}\rangle \\
&\qquad \cup \langle\{\langle C\rangle, \langle B\rangle, b\}, \{(\langle C\rangle, \langle B\rangle), (\langle B\rangle, b)\}\rangle \\
&= \textstyle\bigotimes_{\langle B\rangle}\bigotimes_{\langle C\rangle} \langle\{\langle B\rangle, \langle C\rangle, a, b\}, \{(\langle B\rangle, \langle C\rangle), (\langle C\rangle, a), (\langle C\rangle, \langle B\rangle), (\langle B\rangle, b)\}\rangle \\
&= \textstyle\bigotimes_{\langle B\rangle} \langle\{\langle B\rangle, a, b\}, \{(\langle B\rangle, \langle B\rangle), (\langle B\rangle, a), (\langle B\rangle, b)\}\rangle \\
&= \langle\{a, b\}, \{(\langle B\rangle, \langle B\rangle), (\langle B\rangle, a), (\langle B\rangle, b)\}\rangle
\end{aligned}$$

In the final step, the barrier $\langle B\rangle$ cannot be removed because of the self-loop. So there are two witnesses of the fact that the construction failed: there is still a barrier name in the process, and there is a cycle in the resulting graph.

Theorem 1. *Let P be a process, then P has a deadlock iff $\mathsf{ctg}(P)$ has a cycle. Moreover, if P is deadlock-free (hence it is a DAG) then $(\alpha, \beta) \in \mathsf{ctg}(P)$ iff $\alpha \prec \beta$ (hence the DAG is intransitive).*

Proof (idea). The proof is not difficult but slightly technical. The idea is to extend the notion of execution to go "past" deadlocks, thus detecting cycles in the causal relation. The details are given in companion document. □

In Fig. 2 (left) we describe a system Sys written in the proposed language, together with the covering of $\mathscr{PO}(Sys)$, i.e. its control graph (right). We also indicate the number of its possible executions, a question we address next.

2.2 The counting problem

One may think that in such a simple setting, any behavioral property, such as the counting problem that interests us, could be analyzed efficiently e.g. by a simple induction on the syntax. However, the devil is well hidden inside the box because of the following fact.

Theorem 2. *Let U be a partially ordered set. Then there exists a barrier synchronization process P such that $\mathscr{PO}(P)$ is isomorphic to U.*

$$
\begin{array}{l}
Sys = \mathsf{init}.\nu(G_1,G_2,J_1). \\
\left[\begin{array}{l}
\mathsf{step}_1. \\
\quad \nu(IO) \left[\begin{array}{l}
\mathsf{step}_2.\langle G_1\rangle\mathsf{step}_3. \\
\quad \langle IO\rangle\mathsf{step}_4.\langle G_2\rangle\langle J_1\rangle\mathsf{end} \\
\parallel \mathsf{load}.\mathsf{xform}.\langle IO\rangle 0
\end{array}\right] \\
\parallel \mathsf{gen}.\mathsf{yield}_1.(\langle G_1\rangle 0 \parallel \mathsf{yield}_2.\langle G_2\rangle 0) \\
\parallel \mathsf{fork}.\nu(J_2) \left[\begin{array}{l}
\mathsf{comp}_1.\langle J_2\rangle 0 \\
\parallel \mathsf{comp}_{2.1}.\mathsf{comp}_{2.2}.\langle J_2\rangle 0 \\
\parallel \langle J_2\rangle\mathsf{join}\langle J_1\rangle 0)
\end{array}\right]
\end{array}\right]
\end{array}
$$

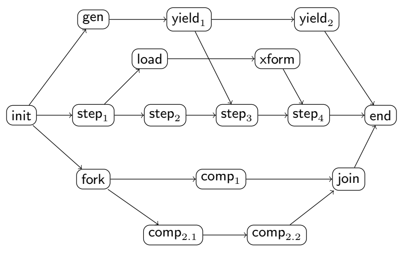
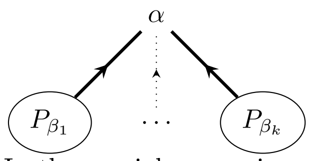


Fig. 2. An example process with barrier synchronizations (left) and its control graph (right). The process is of size 16 and it has exactly 1975974 possible executions.

Proof. (sketch). Consider G the (intransitive) covering DAG of a poset U. We suppose each vertex of G to be uniquely identified by a label ranging over $\alpha_1, \alpha_2, \ldots, \alpha_n$. The objective is to associate to each such vertex labeled α a process expression P_α. The construction is done *backwards*, starting from the *sinks* (vertices without outgoing edges) of G and bubbling-up until its *sources* (vertices without incoming edges).

There is a single rule to apply, considering a vertex labeled α whose children have already been processed, i.e. in a situation depicted as follows:



$$P_\alpha = \langle B_\alpha\rangle\alpha.\,[\langle B_{\beta_1}\rangle 0 \parallel \ldots \parallel \langle B_{\beta_k}\rangle 0]\,.$$

In the special case α is a sink we simply define $P_\alpha = \langle B_\alpha\rangle\alpha.0$. In this construction it is quite obvious that $\alpha \prec \beta_i$ for each of the β_i's, provided the barriers $B_\alpha, B_{\beta_1}, \ldots, B_{\beta_k}$ are defined somewhere in the outer scope.

At the end we have a set of processes $P_{\alpha_1}, \ldots, P_{\alpha_n}$ associated to the vertices of G and we finally define $P = \nu(B_{\alpha_1}) \ldots \nu(B_{\alpha_n})\,[P_{\alpha_1} \parallel \ldots \parallel P_{\alpha_n}]$.

That $\mathscr{PO}(P)$ has the same covering as U is a simple consequence of the construction. □

Corollary 1. *Let P be a non-deadlocked process. Then $\langle\alpha_1, \ldots, \alpha_n\rangle$ is an execution of P if it is a linear extension of $\mathscr{PO}(P)$. Consequently, the number of executions of P is equal to the number of linear extensions of $\mathscr{PO}(P)$.*

We now reach our "negative" result that is the starting point of the rest of the paper: there is no efficient algorithm to count the number of executions, even for such simplistic barrier processes.

Corollary 2. *Counting the number of executions of a (non-deadlocked) barrier synchronization process is $\sharp P$-complete*[5].

[5] A function f is in $\sharp$P if there is a polynomial-time non-deterministic Turing machine M such that for any instance x, $f(x)$ is the number of executions of M that accept x as input. See https://en.wikipedia.org/wiki/%E2%99%AFP-complete

This is a direct consequence of [8] since counting executions of processes boils down to counting linear extensions in (arbitrary) posets.

3 A generic decomposition scheme and its (symbolic) counting algorithm

We describe in this section a generic (and symbolic) solution to the counting problem, based on a systematic decomposition of finite Posets (thus, by Theorem 1, of process expressions) through their covering DAG (i.e. control graphs).

3.1 Decomposition scheme

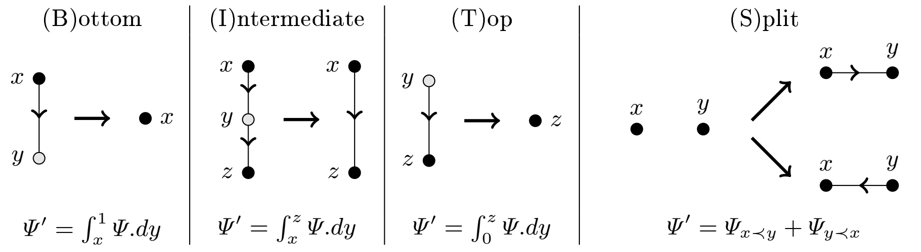

(B)ottom
(I)ntermediate
(T)op
(S)plit
x
y
z
$\Psi' = \int_x^1 \Psi.dy$
$\Psi' = \int_x^z \Psi.dy$
$\Psi' = \int_0^z \Psi.dy$
$\Psi' = \Psi_{x \prec y} + \Psi_{y \prec x}$

Fig. 3. The BITS-decomposition and the construction of the counting formula.

In Fig. 3 we introduce the four decomposition rules that define the BITS-decomposition. The first three rules are somehow straightforward. The (B)-rule (resp. (T)-rule) allows to consume a node with no outgoing (resp. incoming) edge and one incoming (resp. outgoing) edge. In a way, these two rules consume the “pending” parts of the DAG. The (I)-rule allows to consume a node with exactly one incoming and outgoing edge. The final (S)-rule takes two incomparable nodes x, y and decomposes the DAG in two variants: the one for $x \prec y$ and the one for the converse $y \prec x$.

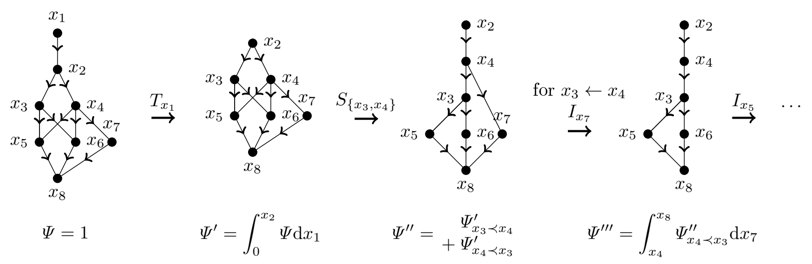
We now discuss the main interest of the decomposition: the incremental construction of an integral formula that solves the counting problem. The calculation is governed by the equations specified below the rules in Fig. 3, in which the current formula Ψ is updated according to the definition of Ψ' in the equations.

Theorem 3. *The numerical evaluation of the integral formula built by the BITS-decomposition yields the number of linear extensions of the corresponding Poset. Moreover, the applications of the BITS-rules are confluent, in the sense that all the sequences of (valid) rules reduce the DAG to an empty graph*[6].

[6] At the end of the decomposition, the DAG is in fact reduced to a single node, which is removed by an integration between 0 and 1.

The precise justification of the integral computation and the proof for the theorem above are postponed to Section 3.2 below. We first consider an example.

Example 1. Illustrating the BITS-decomposition scheme.


T_{x_1}
$S_{\{x_3,x_4\}}$
for $x_3 \leftarrow x_4$
I_{x_7}
I_{x_5}
$\Psi = 1$
$\Psi' = \int_0^{x_2} \Psi \mathrm{d}x_1$
$\Psi'' = \Psi'_{x_3 \prec x_4} + \Psi'_{x_4 \prec x_3}$
$\Psi''' = \int_{x_4}^{x_8} \Psi''_{x_4 \prec x_3} \mathrm{d}x_7$

The DAG to decompose (on the left) is of size 8 with nodes $x_1, \ldots, x_8$. The decomposition is non-deterministic, multiple rules apply, e.g. we could "consume" the node x_7 with the (I) rule. Also, the (S)plit rule is always enabled. In the example, we decide to first remove the node x_1 by an application of the (T) rule. We then show an application of the (S)plit rule for the incomparable nodes x_3 and x_4. The decomposition should then be performed on two distinct DAGs: one for $x_3 \prec x_4$ and the other one for $x_4 \prec x_3$. We illustrate the second choice, and we further eliminate the nodes x_7 then x_5 using the (I) rule, etc. Ultimately all the DAGs are decomposed and we obtain the following integral computation:

$$\Psi = \int_{x_2=0}^{1} \int_{x_4=x_2}^{1} \int_{x_3=x_4}^{1} \int_{x_6=x_3}^{1} \int_{x_8=x_6}^{1} \int_{x_5=x_3}^{x_8} \int_{x_7=x_4}^{x_8} \left(\mathbb{1}_{|x_4 \prec x_3} \cdot \int_{x_1=0}^{x_2} 1 \cdot \mathrm{d}x_1 + \mathbb{1}_{|x_3 \prec x_4} \cdot \int_{x_1=0}^{x_2} 1 \cdot \mathrm{d}x_1 \right) \mathrm{d}x_7 \mathrm{d}x_5 \mathrm{d}x_8 \mathrm{d}x_6 \mathrm{d}x_3 \mathrm{d}x_4 \mathrm{d}x_2 = \frac{8+6}{8!}.$$

The result means that there are exactly 14 distinct linear extensions in the example Poset.

3.2 Embedding in the hypercube: the order polytope

The justification of our decomposition scheme is based on the continuous embedding of posets into the hypercube, as investigated in [19].

Definition 6 (order polytope).
Let $P = (E, \prec)$ be a poset of size n. Let C be the unit hypercube defined by $C = \{(x_1, \ldots, x_n) \in \mathbb{R}^n \mid \forall i,\ 0 \leq x_i \leq 1\}$. For each constraint $x_i \prec x_j \in P$ we define the convex subset $S_{i,j} = \{(x_1, \ldots, x_n) \in \mathbb{R}^n \mid x_i \leq x_j\}$, i.e. one of the half spaces obtained by cutting $\mathbb{R}^n$ with the hyperplane $\{(x_1, \ldots, x_n) \in \mathbb{R}^n \mid x_i - x_j = 0\}$. Thus, the order polytope C_P of P is:

$$C_p = \bigcap_{x_i \prec x_j \in P} S_{i,j} \cap C$$

Each linear extension, seen as total orders, can similarly be embedded in the unit hypercube. Then, the order polytopes of the linear extensions of a poset P form a partition of the Poset embedding C_p as illustrated in Figure 4.

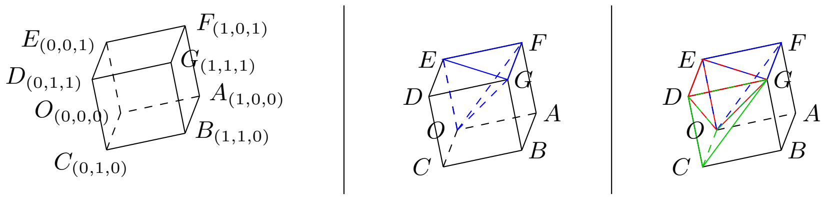


Fig. 4. From left to right: the unit hypercube, the embedding of the total order $1 \prec 2 \prec 3$ and the embedding of the poset $P = (\{1,2,3\}, \{1 \prec 2\})$ divided in its three linear extensions.

The number of linear extensions of a poset P, written $|\mathscr{LE}(P)|$, is then characterized as a volume in the embedding.

Theorem 4. *([19], Corollary 4.2]) Let P be a Poset of size n then its number of linear extensions $|\mathscr{LE}(P)| = n! \cdot Vol(C_P)$ where $Vol(C_P)$ is the volume, defined by the Lebesgue measure, of the order polytope C_P.*

The integral formula introduced in the BITS-decomposition corresponds to the computation of $Vol(C_p)$, hence we may now give the key-ideas of Theorem 3.

Proof (Theorem 3, sketch). We begin with the (S)-rule. Applied on two incomparable elements x and y, the rule partitions the polytope in two regions: one for $x \prec y$ and the other for $y \prec x$. Obviously, the respective volume of the two disjoint regions must be added.
We focus now on the (I)-rule. In the context of Lebesgue integration, the classic Fubini's theorem allows to compute the volume V of a polytope P as an iteration on integrals along each dimension, and this in all possible orders, which gives the confluence property. Thus,

$$V = \int_{[0,1]^n} \mathbb{1}_P(\mathbf{x})\mathrm{d}\mathbf{x} = \int_{[0,1]} \cdots \int_{[0,1]} \mathbb{1}_P((x,y,z,\dots))\mathrm{d}x\mathrm{d}y\mathrm{d}z\dots,$$

$\mathbb{1}_P$ being the indicator function of P such that $\mathbb{1}_P((x,y,z,\dots)) = \prod_{\alpha \text{ actions}} \mathbb{1}_{P_\alpha}(\alpha)$, with P_α the projection of P on the dimension associated to α. By convexity of P, the function $\mathbb{1}_{P_y}$ is the indicator function of a segment $[x,z]$. So the following identity holds: $\int_P \mathbb{1}_{P_y}(y)\mathrm{d}y = \int_x^z \mathrm{d}y$. Finally, the two other rules (T) and (B) are just special cases (taking $x = 0$, alternatively $z = 1$). □

Corollary 3. *(Stanley [19]) The order polytope of a linear extension is a simplex and the simplices of the linear extensions are isometric, thus of the same volume.*

4 Uniform random generation of process executions

In this section we describe a generic algorithm for the uniform random generation of executions of barrier synchronization processes. The algorithm is based on the

BITS-decomposition and its embedding in the unit hypercube. It has two essential properties. First, it is directly working on the control graphs (equivalently on the corresponding poset), and thus does not require the explicit construction of the state-space of processes. Second, it generates possible executions of processes at random according to the uniform distribution. This is a guarantee that the sampling is not biased and reflects the actual behavior of the processes.

Algorithm 1 Uniform sampling of a simplex of the order polytope

```
function SamplePoint[7]($\mathcal{I} = \int_a^b f(y_i)\,\mathrm{d}y_i$)
  $C \leftarrow \mathsf{eval}(\mathcal{I})$   ;   $U \leftarrow$ Uniform$(a, b)$
  $Y_i \leftarrow$ the solution $t$ of $\int_a^t \frac{1}{C} f(y_i)\,\mathrm{d}y_i = U$
  if $f$ is not a symbolic constant then
    SamplePoint($f\{y_i \leftarrow Y_i\}$)
  else return the $Y_i$'s
```

The starting point of Algorithm 1 (cf. previous page) is a Poset over a set of points $\{x_1, \ldots, x_n\}$ (or equivalently its covering DAG). The decomposition scheme of Section 3 produces an integral formula $\mathcal{I}$ of the form $\int_0^1 F(y_n, \ldots, y_1)\,\mathrm{d}y_n \cdots \mathrm{d}y_1$. with F a symbolic integral formula over the points $x_1, \ldots, x_n$. The y variables represent a permutation of the poset points giving the order followed along the decomposition. Thus, the variable y_i corresponds to the i-th removed point during the decomposition. We remind the reader that the evaluation of the formula $\mathcal{I}$ gives the number of linear extensions of the partial order. Now, starting with the complete formula, the variables $y_1,\ y_2, \ldots$ will be eliminated, in turn, in an "outside-in" way. Algorithm 1 takes place at the i-th step of the process. At this step, the considered formula is of the following form:

$$\int_a^b \underbrace{\left(\int \cdots \int 1\ \mathrm{d}y_n \cdots \mathrm{d}y_{i+1}\right)}_{f(y_i)} \mathrm{d}y_i.$$

Note that in the subformula $f(y_i)$ the variable y_i may only occur (possibly multiple times) as an integral bound.

In the algorithm, the variable C gets the result of the numerical computation of the integral $\mathcal{I}$ at the given step. Next we draw (with Uniform) a real number U uniformly at random between the integration bounds a and b. Based on these two intermediate values, we perform a numerical solving of variable t in the integral formula corresponding to the *slice* of the polytope along the hyperplan $y_i = U$. The result, a real number between a and b, is stored in variable Y_i. The justification of this step is further discussed in the proof sketch of Theorem 5 below.

[7] The Python/Sage implementation of the random sampler is available at the following location: https://gitlab.com/ParComb/combinatorics-barrier-synchro/blob/master/code/RandLinExtSage.py

If there remains integrals in $\mathcal{I}$, the algorithm is applied recursively by substituting the variable y_i in the integral bounds of $\mathcal{I}$ by the numerical value Y_i. If no integral remains, all the computed values Y_i's are returned. As illustrated in Example 2 below, this allows to select a specific linear extension in the initial partial ordering. The justification of the algorithm is given by the following theorem.

Theorem 5. *Algorithm 1 uniformly samples a point of the order polytope with a $\mathcal{O}(n)$ complexity in the number of integrations.*

Proof (sketch). The problem is reduced to the uniform random sampling of a point p in the order polytope. This is a classical problem about marginal densities that can be solved by slicing the polytope and evaluating incrementally the n continuous random variables associated to the coordinates of p. More precisely, during the calculation of the volume of the polytope P, the last integration (of a monovariate polynomial $p(y)$) done from 0 to 1 corresponds to integrate the slices of P according the last variable y. So, the polynomial $p(y)/\int_0^1 p(y)dy$ is nothing but the density function of the random variable Y corresponding to the value of y. Thus, we can generate Y according to this density and fix it. When this is done, we can inductively continue with the previous integrations to draw all the random variables associated to the coordinates of p. The linear complexity of Algorithm 1 follows from the fact that each partial integration deletes exactly one variable (which corresponds to one node). Of course at each step a possibly costly computation of the counting formula is required. $\square$

We now illustrate the sampling process based on Example 1 (page 9).

Example 2. First we assume that the whole integral formula has already been computed. To simplify the presentation we only consider (S)plit-free DAGs *i.e.* decomposable without the (S) rule. Note that it would be easy to deal with the (S)plit rule: it is sufficient to uniformly choose one of the DAG processed by the (S)-rule w.r.t. their number of linear extensions.

Thus we will run the example on the DAG of Example 1 where the DAG corresponding to "$x_4 \prec x_3$" as been randomly chosen (with probability $\frac{8}{14}$) *i.e.* the following formula holds:

$$\int_0^1 \left(\int_{x_2}^1 \int_{x_4}^1 \int_{x_3}^1 \int_{x_6}^1 \int_{x_4}^{x_8} \int_{x_3}^{x_8} \int_0^{x_2} \mathrm{d}x_1 \mathrm{d}x_5 \mathrm{d}x_7 \mathrm{d}x_8 \mathrm{d}x_6 \mathrm{d}x_3 \mathrm{d}x_4 \right) \mathrm{d}x_2 = \frac{8}{8!}.$$

In the equation above, the sub-formula between parentheses would be denoted by $f(x_2)$ in the explanation of the algorithm. Now, let us apply the Algorithm 1 to that formula in order to sample a point of the order polytope. In the first step the normalizing constant C is equal to $\frac{8!}{8}$, we draw U uniformly in $[0, 1]$ and so we compute a solution of $\frac{8!}{8} \int_0^t \ldots \mathrm{d}x_2 = U$. That solution corresponds to the second coordinate of a the point we are sampling. And so on, we obtain values for each of the coordinates:

$$\begin{cases} X_1 = 0.064\ldots, & X_2 = 0.081\ldots, & X_3 = 0.541\ldots, & X_4 = 0.323\ldots, \\ X_5 = 0.770\ldots, & X_6 = 0.625\ldots, & X_7 = 0.582\ldots, & X_8 = 0.892\ldots \end{cases}$$

These points belong to a simplex of the order polytope. To find the corresponding linear extension we compute the rank of that vector *i.e.* the order induced by the values of the coordinates correspond to a linear extension of the original DAG:

$$(x_1, x_2, x_4, x_3, x_7, x_6, x_5, x_8).$$

This is ultimately the linear extension returned by the algorithm.

5 Classes of processes that are BIT-decomposable (or not)

Thanks to the BITS decomposition scheme, we can generate a counting formula for any (deadlock-free) process expressed in the barrier synchronization calculus, and derive from it a dedicated uniform random sampler. However the (S)plit rule generates two summands, thus if we cannot find common calculations between the summands the resulting formula can grow exponentially in the size of the concerned process. If we avoid splits in the decomposition, then the counting formula remains of linear size. This is, we think, a good indicator that the subclass of so-called "BIT-decomposable" processes is worth investigating for its own sake. In this Section, we first give some illustrations of the expressivity of this subclass, and we then study the question of what it is to be *not* BIT-decomposable. By lack of space, the discussion in this Section remains rather informal with very rough proof sketches, and more formal developments are left for a future work. Also, the first two subsections are extended results based on previously published papers (respectively [6] and [7]).

5.1 From tree Posets to fork-join parallelism

If the control-graph of a process is decomposed with only the B(ottom) rule (or equivalently the T(op) rule), then it is rather easy to show that its shape is that of a *tree*. These are processes that cannot do much beyond forking sub-processes. For example, based on our language of barrier synchronization it is very easy to encode e.g. the (rooted) binary trees:

$$T ::= 0 \mid \alpha.(T \parallel T) \quad \text{or e.g.} \quad T ::= 0 \mid \nu B \ \ (\alpha.\langle B\rangle 0 \parallel \langle B\rangle T \parallel \langle B\rangle T)$$

The good news is that the combinatorics on trees is well-studied. In the paper [4] we provide a thorough study of such processes, and in particular we describe very efficient counting and uniform random generation algorithms. Of course, this is not a very interesting sub-class in terms of concurrency.

Thankfully, many results on trees generalize rather straightforwardly to *fork-join parallelism*, a sub-class we characterize inductively in Table 1. Informally, this proof system imposes that processes use their synchronization barriers according to a *stack discipline*. When synchronizing, only the last created barrier is available, which exactly corresponds to the traditional notion of a *join* in concurrency. Combinatorially, there is a correspondence between these processes

$$\frac{}{\sigma \vdash_{FJ} 0} \quad \frac{\sigma \vdash_{FJ} P}{\sigma \vdash_{FJ} \alpha.P} \quad \frac{\sigma \vdash_{FJ} P \quad \sigma \vdash_{FJ} Q}{\sigma \vdash_{FJ} P \parallel Q} \quad \frac{B{::}\sigma \vdash_{FJ} P}{\sigma \vdash_{FJ} \nu(B)\ P} \quad \frac{\sigma \vdash_{FJ} P}{B{::}\sigma \vdash_{FJ} \langle B\rangle.P}$$

Table 1. A proof system for fork-join processes.

and the class of *series-parallel Posets*. In the decomposition both the (B) and the (I) rule are needed, but following a tree-structured strategy. Most (if not all) the interesting questions about such partial orders can be answered in (low) polynomial time.

Theorem 6 (cf. [6]). *For a fork join process of size n the counting problem is of time complexity $O(n)$ and we developed a bit-optimal uniform random sampler with time complexity $O(n\sqrt{n})$ on average.*

5.2 Asynchronism with promises

We now discuss another interesting sub-class of processes that can also be characterized inductively on the syntax of our process calculus, but this time using the three BIT-decomposition rules (in a controlled manner). The strict stack discipline of fork-join processes imposes a form of *synchronous* behavior: all the forked processes must terminate before a join may be performed. To support a limited form of *asynchronism*, a basic principle is to introduce *promise* processes.

$$\frac{}{\emptyset \vdash_{ctrl} 0} \quad \frac{\pi \vdash_{ctrl} P}{\pi \vdash_{ctrl} \alpha.P} \quad \frac{\pi \vdash_{ctrl} P}{\pi \cup \{B\} \vdash_{ctrl} \langle B\rangle.P} \quad \frac{B \notin \pi \quad \pi \cup \{B\} \vdash_{ctrl} P \quad Q \uparrow_B}{\pi \vdash_{ctrl} \nu(B)\,(P \parallel Q)}$$

with $Q \uparrow_B$ iff $Q \equiv \alpha.R$ and $R \uparrow_B$ or $Q \equiv \langle B\rangle.0$

Table 2. A proof system for promises.

In Table 2 we define a simple inductive process structure composed as follows. A *main control thread* can perform atomic actions (at any time), and also fork a sub-process of the form $\nu(B)\,(P \parallel Q)$ but with a strong restriction:

- a single barrier B is created for the sub-processes to interact.
- the left sub-process P must be the continuation of the main control thread,
- the right sub-process Q must be a promise, which can only perform a sequence of atomic actions and ultimately synchronize with the control thread.

We are currently investigating this class as a whole, but we already obtained interesting results for the *arch-processes* in [7]. An arch-process follows the constraint of Table 2 but adds further restrictions. The main control thread can still spawn an arbitrary number of promises, however there must be two separate phases for the synchronization. After the first promise synchronizes, the main control thread cannot spawn any new promise. In [7] a supplementary constraint is added (for the sake of algorithmic efficiency): each promise must perform exactly one atomic action, and the control thread can only perform actions when all the promises are running. In this paper, we remove this rather artificial constraint considering a larger, and more useful process sub-class.

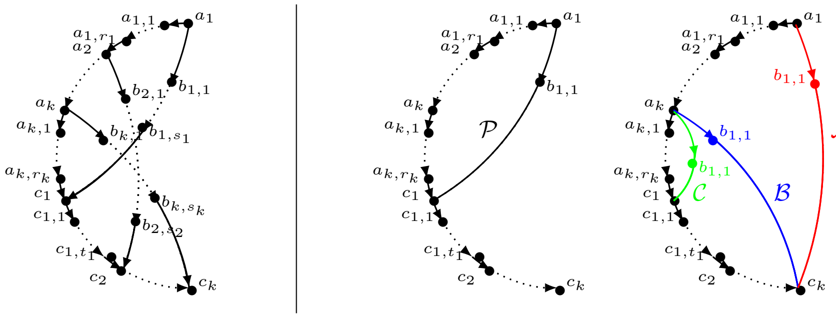


Fig. 5. The structure of an arch-process (left) and the inclusion-exclusion counting principle (right).

In Fig. 5 (left) is represented the general structure of a generalized arch-process. The a_i's actions are the promise forks, and the synchronization points are the c_j's. The constraint is thus that all the a_i's occur before the c_j's.

Theorem 7. *The number of executions of an arch-process can be calculated in $O(n^2)$ arithmetic operations, using a dynamic programming algorithm based on memoization.*

Proof (idea). A complete proof is provided in [7] for "simple" arch-processes, and the generalization is detailed in the companion document. We only describe the *inclusion-exclusion* principle on which our counting algorithm is based. Fig. 5 (right) describes this principles (we omit the representation of the other promises to obtain a clear picture of our approach). Our objective is to count the number of execution contributed by a single promise with atomic action $b_{1,1}$. If we denote by $\ell_{\mathcal{P}}$ this contribution, we reformulate it as a combination $\ell_{\mathcal{P}} = \ell_{\mathcal{A}} - \ell_{\mathcal{B}} + \ell_{\mathcal{C}}$ as depicted on the rightmost part of Fig. 5. First, we take the "virtual" promise $\mathcal{A}$ going from the starting point a_1 of $\ell_{\mathcal{P}}$ until the end point c_k of the main thread. Of course there are two many possibilities if we only keep $\mathcal{A}$. An over-approximation of what it is to remove is the promise $\mathcal{B}$ going from the start of the last promise (at point a_k) until the end. But this time we removed too many possibilities, which corresponds to promise $\mathcal{C}$. The latter is thus reinserted in the count. Each of these three "virtual" promises have a simpler counting procedure. To guarantee the quadratic worst-time complexity (in the number of arithmetic operations), we have to memoize the intermediate results. We refer to the companion document for further details. □

From this counting procedure we developed a uniform random sampler following the principles of the *recursive method*, as described in [10].

Theorem 8. *Let $\mathcal{P}$ be a promise-process of size n with $k \geq n$ promises. A random sampler of $O(n^4)$ time-complexity (in the number of arithmetic operations) builds uniform executions.*

The algorithm and the complete proof are detailed in the companion document. One notable aspect is that in order to get rid of the forbidden case of executions associated to the "virtual" promise $\mathcal{B}$ we cannot only do rejection (because the induced complexity would be exponential). In the generalization of arch-processes, we proceed by case analysis: for each possibility for the insertion of $b_{1,1}$ in the main control thread we compute the relative probability for the associated process $\mathcal{P}$. This explains the increase of complexity (from $O(n^2)$ to $O(n^4)$) if compared to [7].

5.3 BIT-free processes

The class of BIT-decomposable processes is rather large, and we in fact only uncovered two interesting sub-classes that can be easily captured inductively on the process syntax. The relatively non-trivial process Sys of Fig. 2 is also interestingly BIT-decomposable. We now adopt the complementary view of trying to understand the combinatorial structure of a so called "BIT-free" process, which is *not* decomposable using only the (B), (I) and (T) rules.

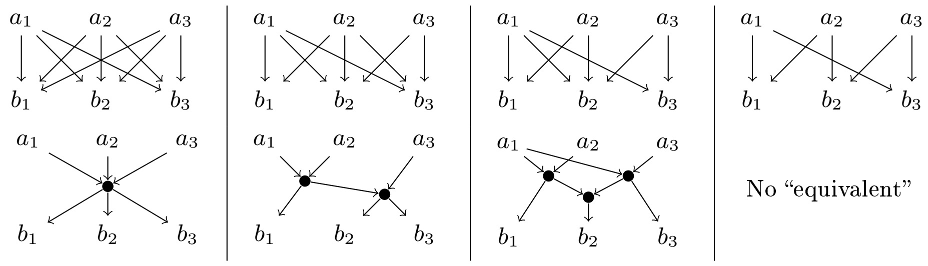


Fig. 6. Typical BIT-free substructures, and their BIT "equivalent" (when possible).

The BIT-free condition implies the occurrence of structures similar to the ones depicted on Fig. 6. These structures are composed of a set of "bottom" processes (the b_i's) waiting for "top" processes (the a_j') according to some synchronization pattern. We represent the whole possibilities of size 3 (up-to order-isomorphism) in the upper-part of the figure. The upper-left process is a complete (directed) bipartite graph, which can in fact be "translated" to a BIT-decomposable process as seen on the lower-part of the figure. This requires the introduction of a single "synchronization point" between the two process groups.

This transformation preserves the number of executions and is Poset-wise equivalent. At each step "to the right" of Fig. 6, we remove a directed edge. In the second and third processes (in the middle), we also have an equivalent with respectively two and three synchronization points. In these cases, the number of linear extensions is not preserved but the "nature" of the order is respected: the interleavings of the initial atomic actions are the same. The only non-transformable structure, let's say the one "truly" BIT-free is the rightmost process. Even if we introduce synchronization points (we need at least three of them), the structure would not become BIT-decomposable. In terms of order theory such a structure is called a *Crown* poset. In [9] it is shown that the counting problem is already ♯-P complete for partial orders of height 2, hence directed bipartite digraphs similar to the structures of Fig. 6. One might wonder if this is still the case when these structures cannot occur, especially in the case of BIT-decomposable processes. This is for us a very interesting (and open) problem.

6 Experimental study

Algorithm	**Class**	**Count.**	**Unif. Rand. Gen.**	**Reference**
FJ	Fork-join	$O(n)$	$O(n \cdot \sqrt{n})$ on average	[6]
Arch	Arch-processes	$O(n^2)$	$O(n^4)$ worst case	[7]/Theorem 8
bit	BIT-decomposable	?	?	Theorem 3
cftp[8]	All processes	–	$O(n^3 \cdot log\ n)$ expected	[14]

Table 3. Summary of counting and uniform random sampling algorithms (time complexity figures with n: number of atomic actions).

In this section, we put into use the various algorihms for counting and generating process executions uniformly at random. Table 3 summarizes these algorithms and the associated worst-case time complexity bounds (when known). We implemented all the algorithms in Python 3, and we did not optimize for efficiency, hence the numbers we obtain only give a rough idea of their performances. For the sake of reproducibility, the whole experimental setting is available in the companion repository, with explanations about the required dependencies and usage. The computer we used to perform the benchmark is a standard laptop PC with an I7-8550U CPU, 8Gb RAM running Manjaro Linux. As an initial experiment, the example of Fig. 2 is BIT-decomposable, so we can apply the bit and cftp algorithms. The counting (of its 1975974 possible executions) takes about 0.3s and it takes about 9 millisecond to uniformly generate an execution with the bit sampler, and about 0.2s with cftp. For "small" state spaces, we observe that bit is always faster than cftp.

[8] The cftp algorithm is the only one we did not design, but only implement. Its complexity is $O(n^3 \cdot log\ n)$ (randomized) expected time.

FJ size	$\sharp\mathscr{LE}$	FJ **gen**	**(count)**	BIT **gen**	**(count)**	CFTP **gen**
10	19	0.00001 s	(0.0002 s)	0.0006 s	(0.03 s)	0.04 s
30	10^9	0.00002 s	(0.0002 s)	0.02 s	(0.03 s)	1.8 s
40	$6 \cdot 10^6$	0.00004 s	(0.0003 s)	3.5 s	(5.2 s)	5.6 s
63	$4 \cdot 10^{29}$	0.0005 s	(0.03 s)	Mem. crash	(Crash)	55 s
217028	$2 \cdot 10^{292431}$	8.11 s	(3.34 s)	Mem. crash	(Crash)	Timeout

Arch size	$\sharp\mathscr{LE}$	ARCH **gen**	**(count)**	BIT **gen**	**(count)**	CFTP **gen**
10:2	43	0.00002 s	(0.00004 s)	0.002 s	(0.000006 s)	0.04 s
30:2	$9.8 \cdot 10^8$	0.003 s	(0.0009 s)	0.000007 s	(0.0004 s)	1.5 s
30:4	$6.9 \cdot 10^{10}$	0.001 s	(0.005 s)	0.000007 s	(0.004 s)	2.5 s
100:2	$1.3 \cdot 10^{32}$	0.75 s	(0.16 s)	Mem. crash	(Crash)	[6] 5.6 s
100:32	$1 \cdot 10^{53}$	2.7 s	(0.17 s)	Mem. crash	(Crash)	[6] 5.9 s
200:66	10^{130}	54 s	(31 s)	Mem. crash	(Crash)	Timeout

Table 4. Benchmark results for BIT-decomposable classes: FJ and Arch.

For a more thorough comparison of the various algorithms, we generated random processes (uniformly at random among all processes of the same size) in the classes of fork-join (FJ) and arch-processes as discussed in Section 5, using our own Arbogen tool[9] or an ad hoc algorithm for arch-processes (presented in the companion repository). For the fork-join structures, the size is simply the number of atomic actions in the process. It is not a surprise that the dedicated algorithms we developed in [6] outperforms the other algorithms by a large margin. In a few second it can handle extremely large state spaces, which is due to the large "branching factor" of the process "forks". The arch-processes represent a more complex structure, thus the numbers are less "impressive" than in the FJ case. To generate the arch-processes (uniformly at random), we used the number of atomic actions as well as the number of spawned promises as main parameters. Hence an arch of size 'n:k' has n atomic actions and k spawned promises. Our dedicated algorithm for arch-process is also rather effective, considering the state-space sizes it can handle. In less than a minute it can generate an execution path uniformly at random for a process of size 200 with 66 spawned promises, the state-space is in the order of 10^{130}. Also, we observe that in all our tests the observable "complexity" is well below $O(n^4)$. The reason is that we perform the pre-computations (corresponding to the worst case) in a just-in-time (JIT) manner, and in practice we only actually need a small fractions of the computed values. However the random sampler is much more efficient with the separate precomputation. As an illustration, for arch-processes of size 100 with 32 arches, the sampler becomes about 500 times faster. However the memory requirement

[8] For arch-processes of size 100 with 2 arches or 32, the CFTP algorithm timeouts (30s) for almost all of the input graphs.

[9] Arbogen is uniform random generation for context-free grammar structures: cf. https://github.com/fredokun/arbogen.

for the precomputation grows very quickly, so that the JIT variant is clearly preferable.

In both the FJ and arch-process cases the current implementation of the BIT algorithms is not entirely satisfying. One reason is that the strategy we employ for the BIT-decomposition is quite "oblivious" to the actual structure of the DAG. As an example, this strategy handles fork-joins far better than arch-processes. In comparison, the CFTP algorithm is less sensitive to the structure, it performs quite uniformly on the whole benchmark. We are still confident that by handling the integral computation natively, the BIT algorithms could handle much larger state-spaces. For now, they are only usable up-to a size of about 40 nodes (already corresponding to a rather large state space).

7 Conclusion and future work

The process calculus presented in this paper is quite limited in terms of expressivity. In fact, as the paper makes clear it can only be used to describe (intransitive) directed acyclic graphs! However we still believe it is an interesting "core synchronization calculus", providing the minimum set of features so that processes are isomorphic to the whole combinatorial class of partially ordered sets. Of course, to become of any practical use, the barrier synchronization calculus should be complemented with e.g. non-deterministic choice (as we investigate in [4]). Moreover, the extension of our approach to iterative processes remains full of largely open questions.

Another interest of the proposed language is that it can be used to define process (hence poset) sub-classes in an inductive way. We give two illustrations in the paper with the *fork-join* processes and *promises*. This is complementary to definitions wrt. some combinatorial properties, such as the "BIT-decomposable" vs. "BIT-free" sub-classes. The class of arch-processes (that we study in [7] and generalize in the present paper) is also interesting: it is a combinatorially-defined sub-class of the inductively-defined asynchronous processes with promises. We see as quite enlightening the meeting of these two distinct points of view.

Even for the "simple" barrier synchronizations, our study is far from being finished because we are, in a way, also looking for "negative" results. The counting problem is hard, which is of course tightly related to the infamous "combinatorial explosion" phenomenon in concurrency. We in fact believe that the problem remains intractable for the class of BIT-decomposable processes, but this is still an open question that we intend to investigate furthermore. By delimiting more precisely the "hardness" frontier, we hope to find more interesting sub-classes for which we can develop efficient counting and random sampling algorithms.

Acknowledgment We thank the anonymous reviewers as well as our "shepard" for helping us making the paper better and hopefully with less errors.

References

1. Abbes, S., Mairesse, J.: Uniform generation in trace monoids. In: MFCS 2015. LNCS, vol. 9234, pp. 63–75. Springer (2015)
2. Banderier, C., Marchal, P., Wallner, M.: Rectangular Young tableaux with local decreases and the density method for uniform random generation (short version). In: GASCom 2018. Athens, Greece (Jun 2018)
3. Basset, N., Mairesse, J., Soria, M.: Uniform sampling for networks of automata. In: Concur 2017. LIPIcs, vol. 85, pp. 36:1–36:16. Schloss Dagstuhl (2017)
4. Bodini, O., Genitrini, A., Peschanski, F.: The combinatorics of non-determinism. In: FSTTCS'13. LIPIcs, vol. 24, pp. 425–436. Schloss Dagstuhl (2013)
5. Bodini, O., Genitrini, A., Peschanski, F.: A Quantitative Study of Pure Parallel Processes. Electronic Journal of Combinatorics **23**(1), P1.11, 39 pages (2016)
6. Bodini, O., Dien, M., Genitrini, A., Peschanski, F.: Entropic uniform sampling of linear extensions in series-parallel posets. In: CSR 2017. LNCS, vol. 10304, pp. 71–84. Springer (2017)
7. Bodini, O., Dien, M., Genitrini, A., Viola, A.: Beyond series-parallel concurrent systems: The case of arch processes. In: Analysis of Algorithms, AofA 2018. LIPIcs, vol. 110, pp. 14:1–14:14 (2018)
8. Brightwell, G., Winkler, P.: Counting linear extensions is #P-complete. In: STOC. pp. 175–181 (1991)
9. Dittmer, S., Pak, I.: Counting linear extensions of restricted posets. arXiv e-prints arXiv:1802.06312 (Feb 2018)
10. Flajolet, P., Zimmermann, P., Cutsem, B.V.: A calculus for the random generation of labelled combinatorial structures. Theor. Comput. Sci. **132**(2), 1–35 (1994)
11. Gerbessiotis, A.V., Valiant, L.G.: Direct bulk-synchronous parallel algorithms. J. Parallel Distrib. Comput. **22**(2), 251–267 (1994)
12. Grosu, R., Smolka, S.A.: Monte carlo model checking. In: TACAS'05. LNCS, vol. 3440, pp. 271–286. Springer (2005)
13. Hensgen, D., Finkel, R.A., Manber, U.: Two algorithms for barrier synchronization. International Journal of Parallel Programming **17**(1), 1–17 (1988)
14. Huber, M.: Fast perfect sampling from linear extensions. Discrete Mathematics **306**(4), 420–428 (2006)
15. Liskov, B., Shrira, L.: Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In: PLDI'88. pp. 260–267. ACM (1988)
16. Oudinet, J., Denise, A., Gaudel, M., Lassaigne, R., Peyronnet, S.: Uniform monte-carlo model checking. In: FASE 2011. LNCS, vol. 6603. Springer (2011)
17. Rival, I. (ed.): Algorithms and Order. NATO Science Series, Springer (1988)
18. Sen, K.: Effective random testing of concurrent programs. In: Automated Software Engineering ASE'07. pp. 323–332. ACM (2007)
19. Stanley, R.P.: Two poset polytopes. Discrete & Computational Geometry **1**, 9–23 (1986)