



HAL
open science

Quantitative and Algorithmic aspects of Barrier Synchronization in Concurrency

Olivier Bodini, Matthieu Dien, Antoine Genitrini, Frédéric Peschanski

► **To cite this version:**

Olivier Bodini, Matthieu Dien, Antoine Genitrini, Frédéric Peschanski. Quantitative and Algorithmic aspects of Barrier Synchronization in Concurrency. 2019. hal-02301925v1

HAL Id: hal-02301925

<https://hal.science/hal-02301925v1>

Preprint submitted on 30 Sep 2019 (v1), last revised 20 Jan 2021 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Quantitative and Algorithmic aspects of Barrier Synchronization in Concurrency*

Olivier Bodini¹ Matthieu Dien² Antoine Genitrini³ Frédéric Peschanski³

¹ Université Paris-Nord – LIPN – CNRS UMR 7030

² Université de Caen – GREYC – CNRS UMR 6072

³ Sorbonne University – LIP6 – CNRS UMR 7607

received 2019-09-30,

In this paper we address the problem of understanding Concurrency Theory from a combinatorial point of view. We are interested in quantitative results and algorithmic tools to refine our understanding of the classical combinatorial explosion phenomenon arising in concurrency. This paper is essentially focusing on the 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, Partial Order Theory.

1 Introduction

The objective of our research project is to study the *combinatorics* of concurrent processes. Because of the constraints induced by the combinatorics modelization we study *process calculi* with a restricted focus based on some incremental expressivity for the models. For example in [BGP16] the processes we study can only perform atomic actions and fork child processes. In [BGP13] 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 [HFM88].

We propose here to enumerate the number of executions of processes with respect to their syntactic size. 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

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

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 sample 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, we isolate interesting sub-classes wrt. the counting and random sampling problem. We identify well-known structural sub-classes such as *fork-join parallelism* [GV94] and *asynchronous processes with promises* [LS88]. An important property of our decomposition relies in the fact that the order of application of the rules is confluent. Thus, whatever the strategy we follow the global result is the same (and correct). But as we will see, for some strategies, the calculations are easier in practice and the approaches are thus more efficient. In particular for some of these sub-classes we develop dedicated and efficient counting and random sampling algorithms: once the strategy is well understood, we further can simplify the decomposition in order to exhibit algorithms that not really removes nodes one by one. 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).

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 for example in [AM15]) 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 [BMS17]. 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 [AM15] 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-off 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. An uniform random sampler for lassos is proposed in [ODG⁺11]. 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 [Hub06]. The approach we propose, based on the continuous embedding of partial order sets into the hypercube, is quite complementary. A similar idea is used in [BMW18] for the enumeration of Young tableaux using what is there called the *density method*. The

paper [GS05] advocates the uniform random generation of executions as an important building block for *statistical model-checking*. A similar discussion is proposed in [Sen07] for *random testing*. The *leit-motiv* in both cases is that generating execution paths *without* any bias is difficult. Hence an 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 [Riv88] orchestrated by Ivan Rival in late 1980's only with powerful new tools available in the modern combinatorics toolbox.

Outline of the paper

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 we provide the full source code developed in the realm of this work, as well as the benchmark scripts. All these complement information are available online⁽ⁱ⁾. This paper is an updated and extended version of papers [BDGP19] and [BDGP17b]. It contains new material, especially the study of the interesting process sub-classes. The proofs in this extended version are also more detailed.

2 Modelization of a process algebra

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

Definition 2.1 (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:

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

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*. The different constructors are explained on the following very basic process seen as an example:

$$\nu(B) [a_1.\langle B \rangle a_2.0 \parallel \langle B \rangle b_1.0 \parallel c_1.\langle B \rangle 0].$$

This process starts with a barrier, named B , that is broadcasted. Then the 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

⁽ⁱ⁾ cf. <https://gitlab.com/ParComb/combinatorics-barrier-synchro.git>

synchronize on barrier B (it was not the case before thus the synchronization could not take place earlier):

$$\nu(B) [\langle B \rangle a_2.0 \parallel \langle B \rangle b_1.0 \parallel \langle B \rangle 0].$$

The possible next transitions are either $\xrightarrow{a_2} b_1.0 \xrightarrow{b_1} 0$, or, alternatively, $\xrightarrow{b_1} a_2.0 \xrightarrow{a_2} 0$. In the resulting states, the barrier B has been “consumed”.

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

Definition 2.2 (Operational semantics). The operational semantics related to the process language is the following :

$$\begin{array}{c} \frac{}{\alpha.P \xrightarrow{\alpha} P} \text{ (act)} \quad \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \text{ (lpar)} \quad \frac{Q \xrightarrow{\alpha} Q'}{P \parallel Q \xrightarrow{\alpha} P \parallel Q'} \text{ (rpar)} \\ \\ \frac{\text{sync}_B(P)=Q \quad \text{wait}_B(Q) \quad P \xrightarrow{\alpha} P'}{\nu(B)P \xrightarrow{\alpha} \nu(B)P'} \text{ (lift)} \quad \frac{\text{sync}_B(P)=Q \quad \neg \text{wait}_B(Q) \quad Q \xrightarrow{\alpha} Q'}{\nu(B)P \xrightarrow{\alpha} Q'} \text{ (sync)} \end{array}$$

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

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 $\text{sync}_B(P)$ 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 $\text{wait}_B(Q)$ 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 an execution of a process.

Definition 2.3 (Execution). An *execution* σ of a process P is a finite sequence $\langle \alpha_1, \dots, \alpha_n \rangle$ such that there exist a set of processes $P'_{\alpha_1}, \dots, P'_{\alpha_n}$ and a path $P \xrightarrow{\alpha_1} P'_{\alpha_1} \dots \xrightarrow{\alpha_n} P'_{\alpha_n}$ with $P'_{\alpha_n} \rightarrow$ (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, $\alpha_1, \dots, \alpha_n$. This is allowed since the actions are uninterpreted in the semantics (cf. Definition 2.2). Thus, each action α in an execution σ can be associated to a unique *position*, which we denote by $\sigma(\alpha)$. For example if $\sigma = \langle \alpha_1, \dots, \alpha_k, \dots, \alpha_n \rangle$, then $\sigma(\alpha_k) = k$.

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

Definition 2.4 (Cause, direct cause). Let P be a process. An action α of P is said a *cause* of another action β , denoted by $\alpha < \beta$, if and only if for any execution σ of P we have $\sigma(\alpha) < \sigma(\beta)$. Moreover, α is a *direct cause* of β , denoted by $\alpha < \beta$ if and only if $\alpha < \beta$ and there is no γ such that $\alpha < \gamma < \beta$. The relation $<$ obtained from P is denoted by $\mathcal{PO}(P)$.

Obviously $\mathcal{PO}(P)$ is a *partially ordered set* (poset) with covering $<$, 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 $\mathcal{PO}(P)$ 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).

$$\nu(B) [\alpha_1.\langle B \rangle \parallel \alpha_2.\langle B \rangle \parallel \dots \parallel \alpha_n.\langle B \rangle \parallel \langle B \rangle.\beta_1 \parallel \langle B \rangle.\beta_2 \parallel \dots \parallel \langle B \rangle.\beta_n]$$

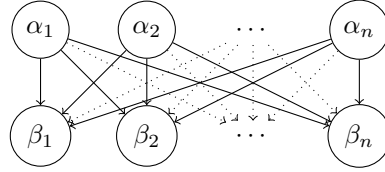


Fig. 1: A process of size $2n$ and its control graph with $2n$ nodes and n^2 edges.

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 directly construct this control graph, according to the following definition.

Definition 2.5 (Construction of control graphs). Let P be a process term. Its control graph is $\text{ctg}(P) = \langle V, E \rangle$, constructed inductively as follows:

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

$$\text{with} \left\{ \begin{array}{l} x \rightsquigarrow \langle V, E \rangle = \langle V \cup \{x\}, \{(x, y) \mid y \in \text{sources}(E) \vee (E = \emptyset \wedge y \in V)\} \rangle \\ \text{sources}(E) = \{y \mid (y, z) \in E \wedge \nexists x, (x, y) \in E\} \\ \otimes_{\langle B \rangle} \langle V, E \rangle = \langle V \setminus \{\langle B \rangle\}, E \setminus \{(x, y) \mid x \neq y \wedge (x = \langle B \rangle \vee y = \langle B \rangle)\} \\ \quad \cup \{(\alpha, \beta) \mid \{(\alpha, \langle B \rangle), (\langle B \rangle, \beta)\} \subseteq E\} \rangle. \end{array} \right.$$

Given a control graph Γ , the notation $x \rightsquigarrow \Gamma$ corresponds to prefixing the graph by a single atomic action. The set $\text{sources}(E)$ corresponds to the *sources* of the edges in E , i.e. the vertices without an incoming edge. And $\otimes_{\langle B \rangle} \Gamma$ 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:

$$\begin{aligned}
\text{ctg}(\nu(B)\nu(C)[\langle B \rangle \langle C \rangle a.0 \mid \langle B \rangle \langle C \rangle b.0]) &= \bigotimes_{\langle B \rangle \langle C \rangle} (\text{ctg}(\langle B \rangle \langle C \rangle a.0) \cup \text{ctg}(\langle B \rangle \langle C \rangle b.0)) \\
&= \bigotimes_{\langle B \rangle \langle C \rangle} \langle \{ \langle B \rangle, \langle C \rangle, a \}, \{ \langle B \rangle, \langle C \rangle \}, \langle \langle C \rangle, a \rangle \} \rangle \\
&\quad \cup \langle \{ \langle B \rangle, \langle C \rangle, b \}, \{ \langle B \rangle, \langle C \rangle \}, \langle \langle C \rangle, b \rangle \} \rangle \\
&= \bigotimes_{\langle B \rangle \langle C \rangle} \langle \{ \langle B \rangle, \langle C \rangle, a, b \}, \{ \langle B \rangle, \langle C \rangle \}, \langle \langle C \rangle, a \rangle, \langle \langle C \rangle, b \rangle \} \rangle \\
&= \bigotimes_{\langle B \rangle} \langle \{ \langle B \rangle, a, b \}, \{ \langle B \rangle, a \}, \langle \langle B \rangle, b \rangle \} \rangle \\
&= \langle \{ a, b \}, \emptyset \rangle
\end{aligned}$$

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.

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

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 2.1. *Let P be a process, then P has a deadlock if and only if $\text{ctg}(P)$ has a cycle. Moreover, if P is deadlock-free (hence it is a DAG) then $(\alpha, \beta) \in \text{ctg}(P)$ if and only if $\alpha < \beta$ (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 Appendix A not to overload the core of the paper. \square

In Fig. 2 (top) we describe a system Sys written in the proposed language, together with the covering of $\mathcal{PO}(Sys)$, i.e. its control graph (bottom). 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.

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

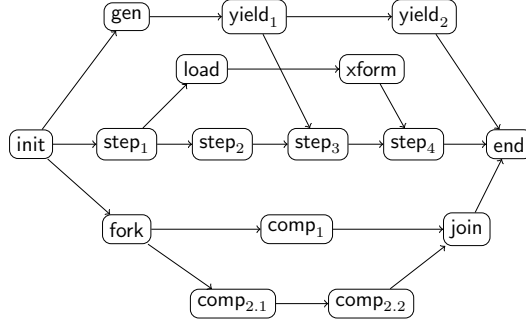
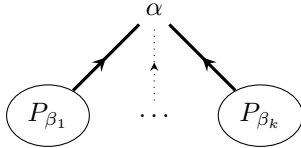


Fig. 2: An example process with barrier synchronizations (top) and its control graph (bottom). The process is of size 16 and it has exactly 1975974 possible executions.

Theorem 2.2. *Let U be a partially ordered set. Then there exists a barrier synchronization process P such that $\mathcal{PO}(P)$ is isomorphic to U .*

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 $\alpha_1, \alpha_2, \dots, \alpha_n$. 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:



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

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 $\alpha < \beta_i$ for each of the β_i 's, provided the barriers $B_\alpha, B_{\beta_1}, \dots, B_{\beta_k}$ are defined somewhere in the outer scope.

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

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

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

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⁽ⁱⁱ⁾.*

This is a direct consequence of [BW91] since counting executions of processes boils down to counting linear extensions in (arbitrary) posets.

3 BITS-Decomposition of a process: shrinking a process to obtain a symbolic enumeration of executions

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 2.1, of process expressions) through their covering DAG (i.e. control graphs).

3.1 Decomposition scheme

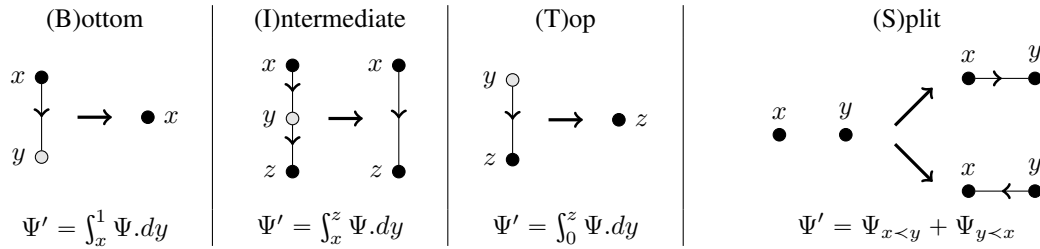


Fig. 3: The BITS-decomposition and the construction of the counting formula.

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 $x < y$ and the one for the converse $y < x$.

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.

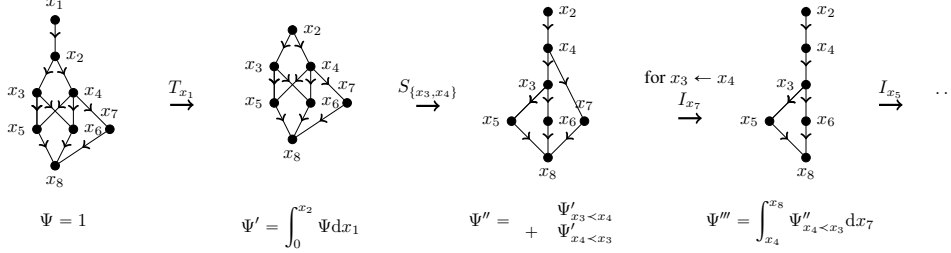
Theorem 3.1. *The numerical evaluation of the integral formula built by the BITS-decomposition yields the number of linear extensions of the corresponding Poset. Moreover, the applications of the BITS-rules*

⁽ⁱⁱ⁾ 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 for example [AB09].

are confluent, in the sense that all the sequences of (valid) rules reduce the DAG to an empty graph⁽ⁱⁱⁱ⁾.

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 3.1. Illustrating the BITS-decomposition scheme.



The DAG to decompose (on the left) is of size 8 with nodes x_1, \dots, 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 $x_3 < x_4$ and the other one for $x_4 < x_3$. 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:

$$\Psi = \int_{x_2=0}^1 \int_{x_4=x_2}^1 \int_{x_3=x_4}^1 \int_{x_6=x_3}^1 \int_{x_8=x_6}^1 \int_{x_5=x_3}^{x_8} \int_{x_7=x_4}^{x_8} \left(\mathbb{1}_{|x_4 < x_3} \cdot \int_{x_1=0}^{x_2} 1 \cdot dx_1 + \mathbb{1}_{|x_3 < x_4} \cdot \int_{x_1=0}^{x_2} 1 \cdot dx_1 \right) dx_7 dx_5 dx_8 dx_6 dx_3 dx_4 dx_2 = \frac{8+6}{8!}.$$

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 [Sta86].

Definition 3.1 (Order polytope). Let $P = (E, <)$ be a poset of size n . Let C be the unit hypercube defined by $C = \{(x_1, \dots, x_n) \in \mathbb{R}^n \mid \forall i, 0 \leq x_i \leq 1\}$. For each constraint $x_i < x_j \in P$ we define the convex subset $S_{i,j} = \{(x_1, \dots, x_n) \in \mathbb{R}^n \mid x_i \leq x_j\}$, i.e. one of the half spaces obtained by cutting \mathbb{R}^n with the hyperplane $\{(x_1, \dots, x_n) \in \mathbb{R}^n \mid x_i - x_j = 0\}$. Thus, the order polytope C_P of P is:

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

Each linear extension, seen as a total order, 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.

⁽ⁱⁱⁱ⁾ 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.

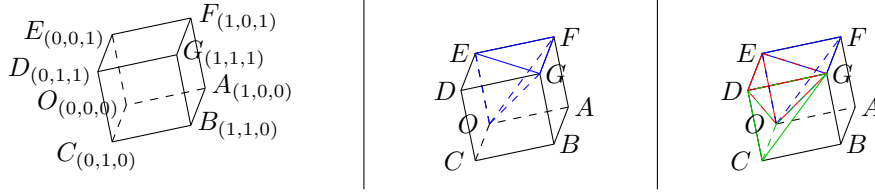


Fig. 4: From left to right: the unit hypercube, the embedding of the total order $1 < 2 < 3$ and the embedding of the poset $P = (\{1, 2, 3\}, \{1 < 2\})$ divided in its three linear extensions.

The number of linear extensions of a Poset P , written $|\mathcal{L}^{\mathcal{E}}(P)|$, is then characterized as a volume in the embedding.

Theorem 3.2. ([Sta86, Corollary 4.2]) *Let P be a Poset of size n then its number of linear extensions $|\mathcal{L}^{\mathcal{E}}(P)| = n! \cdot \text{Vol}(C_P)$ where $\text{Vol}(C_P)$ 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 $\text{Vol}(C_P)$, hence we may now give the key-ideas of Theorem 3.1.

Proof sketch of Theorem 3.1: We begin with the (S)-rule. Applied on two incomparable elements x and y , the rule partitions the polytope in two regions: one for $x < y$ and the other for $y < x$. 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,

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

$\mathbb{1}_P$ 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: $\int_P \mathbb{1}_{P_y}(y) dy = \int_x^z dy$. Finally, the two other rules (T) and (B) are just special cases (taking $x = 0$, alternatively $z = 1$). \square

Corollary 3. ([Sta86]) *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 graph (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

Algorithm 1 Uniform sampling of a simplex of the order polytope

```

function SAMPLEPOINT(iv)( $\mathcal{I} = \int_a^b f(y_i) dy_i$ )
   $C \leftarrow \text{eval}(\mathcal{I})$ 
   $U \leftarrow \text{UNIFORM}(a, b)$ 
   $Y_i \leftarrow$  the solution  $t$  of  $\int_a^t \frac{1}{C} f(y_i) dy_i = U$ 
  if  $f$  is not a symbolic constant then
    SAMPLEPOINT( $f\{y_i \leftarrow Y_i\}$ )
  else return the  $Y_i$ 's

```

uniform distribution. This is a guarantee that the sampling is not biased and reflects the actual behavior of the processes.

The starting point of Algorithm 1 is a Poset over a set of points $\{x_1, \dots, x_n\}$ (or equivalently its covering DAG). The decomposition scheme of Section 3 produces an integral formula \mathcal{I} of the form $\int_0^1 F(y_n, \dots, y_1) dy_n \cdots dy_1$ with F a symbolic integral formula over the points x_1, \dots, 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, \dots 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:

$$\int_a^b \underbrace{\left(\int \cdots \int 1 dy_n \cdots dy_{i+1} \right)}_{f(y_i)} dy_i.$$

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 4.1 below.

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 4.1 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 4.1. *Algorithm 1 uniformly samples a point of the order polytope with a $\mathcal{O}(n)$ complexity in the number of integrations.*

Proof: 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

^(iv) 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>

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 univariate 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 3.1 (page 9).

Example 4.1. 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 3.1 where the DAG corresponding to “ $x_4 < x_3$ ” as been randomly chosen (with probability $\frac{8}{14}$) *i.e.* the following formula holds:

$$\int_0^1 \left(\int_{x_2}^1 \int_{x_4}^1 \int_{x_3}^1 \int_{x_6}^1 \int_{x_4}^{x_8} \int_{x_3}^{x_8} \int_0^{x_2} dx_1 dx_5 dx_7 dx_8 dx_6 dx_3 dx_4 \right) dx_2 = \frac{8}{8!}.$$

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 $\frac{8!}{8} \int_0^t \dots dx_2 = U$. 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:

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

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 Characterization of important process sub-classes and link with BIT-decomposition

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. 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 [BDGP17a] and [BDGV18]).

5.1 From tree Posets to fork-join parallelism

5.1.1 Tree processes

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:

$$T ::= 0 \mid \alpha.(T \parallel T) \quad \text{or e.g.} \quad T ::= 0 \mid \nu B (\alpha.\langle B \rangle 0 \parallel \langle B \rangle T \parallel \langle B \rangle T) \quad (1)$$

The good news is that the combinatorics on trees is well-studied. This study relies on the combinatorial interpretation of processes as discrete structures then the use of tools from the theory of Analytic Combinatorics (see [FS09] for a reference).

The equations (1) are very similar to the combinatorial specification \mathcal{T} of binary trees *i.e.*

$$\mathcal{T} = \mathcal{E} + \mathcal{Z} \times \mathcal{T}^2,$$

which is the way we study syntactic processes.

Concerning the semantic, as mentioned in Corollary 1, executions of a process P correspond to linear extensions of the Poset $\mathcal{PO}(P)$. Another point of view is to consider increasing labelings of the covering DAG which are isomorphic to linear extensions. Hence we can derive from the previous *unlabeled* specification for \mathcal{T} the combinatorial class of binary tree processes, a *labeled* specification for \mathcal{R} the combinatorial class of their runs:

$$\mathcal{R} = \mathcal{E} + \mathcal{Z}^\square \star \mathcal{R}^2.$$

In the paper [BGP13] 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 expressive sub-class in terms of concurrency.

5.1.2 Fork-join processes and Multi Bulk Synchronous Parallel computing (BSP)

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

Fig. 5: A proof system for fork-join processes.

Thankfully, many results on trees generalize rather straightforwardly to *fork-join parallelism*, a subclass we characterize inductively in Fig. 5. Informally, this proof system imposes that processes use their

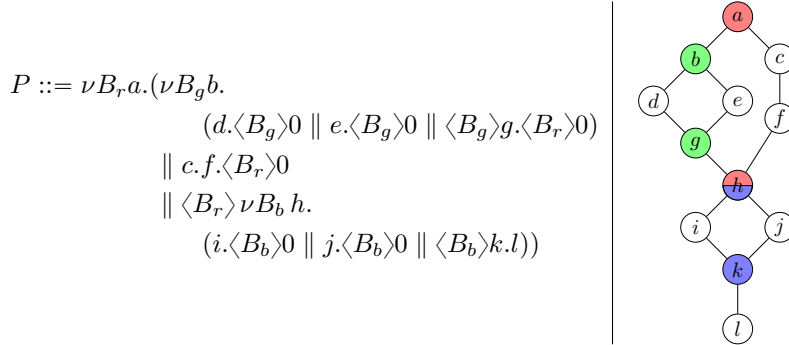


Fig. 6: A fork-join process.

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. The Fig. 6 gives an example of fork-join process P where the colored vertices correspond to “forks” and their relatives “joins”. Like for binary tree processes we can design a combinatorial specification of the combinatorial class \mathcal{F} of fork-join processes:

$$\mathcal{F} = \mathcal{E} + \mathcal{Z} \times \mathcal{F} + \mathcal{Z} \times \mathcal{F}^2 \times \mathcal{F}.$$

Let us explain this specification from the proof system of Fig. 5. The first term \mathcal{E} corresponds to the axiom (the leftmost rule) of Fig. 5; the second term $\mathcal{Z} \times \mathcal{F}$ corresponds to the processes prefixed by an action; the last term $\mathcal{Z} \times \mathcal{F}^2 \times \mathcal{F}$ corresponds to processes composed of two parallel processes (third rule) prefixed by a barrier declaration (B added in the stack β in the fourth rule) and such that the next barrier reached should have the same name as the last barrier stacked (fifth rule).

That computation model is more realistic than the tree processes. Actually, the *Multi Bulk Synchronous Parallel* (Multi-BSP) model of computations (see the seminal paper [Val11]) can be seen as a fork-join model of computations. The Multi-BSP model defines a tree of nested computational components: the leaves are the processors and the inner vertices are computers and more. For example, a height 4 tree would be a data center (the root of the tree), composed of server racks (depth 1), each composed of servers (depth 2) with several multi-core processors (depth 3). Then the Multi-BSP model sets that each vertex obey to the original BSP model. The BSP model states that processing units computations are divided in *superstep* composed of (asynchronous) computations, communications requests (between processing units) and ending by a barrier synchronization during which the communications are processed. So supersteps at depth i correspond to fork-join processes where i barriers names are visible, put another way it corresponds to sub-DAG of depth i from the root.

5.1.3 The ordered product

Like in the case of binary tree processes we can derive the class of increasingly labeled fork-join processes corresponding to their runs. But unlike the previous case, the boxed product is not expressive enough to give a specification of such increasingly labeled class. Here we need a global constraint over the labels

such that the labels of the upper part (corresponding to the $z \times \mathcal{F}^2$ term) are smaller than the one of the bottom part of the Poset (the last \mathcal{F} term). That is the purpose of the *ordered product*, introduced in the context of *species theory* (see [BLL98]), that we studied with an analytic combinatorics point of view in [BDGP17b].

Definition 5.1. Let \mathcal{A} and \mathcal{B} be two labeled combinatorial classes and α and β be two structures respectively in \mathcal{A} and in \mathcal{B} . We define the class of labeled structures induced by α and β :

$$\alpha \boxtimes \beta = \{(\alpha, f_{|\alpha|}(\beta)) \mid f_{|\alpha|}(\cdot) \text{ shifts the labels from } \beta \text{ by } |\alpha|\},$$

such that the function $f_{|\alpha|}$ is a relabeling function (by shifting by $+|\alpha|$ the previous labels).

We extend the ordered product to combinatorial classes:

$$\mathcal{A} \boxtimes \mathcal{B} = \bigcup_{\alpha \in \mathcal{A}, \beta \in \mathcal{B}} \alpha \boxtimes \beta.$$

In fact, the ordered product of $\mathcal{A} \boxtimes \mathcal{B}$ contains objects from the product $\mathcal{A} \star \mathcal{B}$ such that all the labels of component of \mathcal{A} are smaller than the ones of the component of \mathcal{B} .

As usual, this operator over combinatorial classes translates into an operator over generating functions. Before introducing that translation we first recall the classical integral transforms: the combinatorial Laplace and the Borel transforms^(v). From a combinatorial point of view, they define a bridge between exponential generating functions and ordinary generating functions. More precisely, we have respectively

$$\mathcal{L}_c \left(\sum_{n \geq 0} a_n \frac{z^n}{n!} \right) = \sum_{n \geq 0} a_n z^n; \quad \mathcal{B}_c \left(\sum_{n \geq 0} a_n z^n \right) = \sum_{n \geq 0} a_n \frac{z^n}{n!}.$$

From a functional point of view, the combinatorial Laplace and the Borel transforms correspond respectively to

$$\begin{aligned} \mathcal{L}_c(f) &= \int_0^\infty \exp(-t) f(zt) dt; \\ \mathcal{B}_c(f) &= \frac{1}{2i\pi} \int_{c-i\infty}^{c+i\infty} \frac{\exp(zt)}{t} f\left(\frac{1}{t}\right) dt, \end{aligned}$$

where the real constant c is greater than the real part of all singularities of $f(1/t)/t$.

Analogously to the traditional Laplace transform, the product of Laplace transforms can be expressed with a convolution product:

$$z \cdot \mathcal{L}_c(f) \cdot \mathcal{L}_c(g) = \mathcal{L}_c \left(\int_0^z f(t)g(z-t) dt \right).$$

Equivalently

$$\mathcal{L}_c(f) \cdot \mathcal{L}_c(g) = \mathcal{L}_c \left(\int_0^z f(t)g'(z-t) dt + g(0)f(z) \right).$$

We denote by $f * g$ the combinatorial convolution $\int_0^z f(t)g'(z-t) dt + g(0)f(z)$.

^(v) cf. Appendix B in which we recall the relations between the classical Laplace and Borel transforms and their combinatorial definitions.

Proposition 5.1. *Let \mathcal{A} and \mathcal{B} be two labeled combinatorial classes. The exponential generating function $C(z)$, associated to $\mathcal{C} = \mathcal{A} \boxtimes \mathcal{B}$, satisfies the three following equations (according to the context: formal or integrable functions)*

$$\begin{aligned} C(z) &= \mathcal{B}_c(\mathcal{L}_c A(z) \cdot \mathcal{L}_c B(z)) \\ &= \sum_{n \geq 0} \frac{\sum_{k=0}^n a_k b_{n-k}}{n!} z^n \\ &= A(z) * B(z). \end{aligned}$$

The proof of the result is given in Appendix B.

Observe that the ordered product gives a combinatorial interpretation of this adapted convolution. Note that the integral interpretation is valid when both generating function $A(z)$ and $B(z)$ are integrable in their definition domain. However, for example if $A(z) = 1/(1-z)$, although $\mathcal{L}_c A(z)$ is not analytic, the function $A(z)$ can be a component of the ordered product.

5.1.4 Combinatorics of fork-join processes

The introduction of the ordered product allows us to define several classes of increasingly labelled fork-join processes with different constraints. Here we focus on the class \mathcal{F}_ℓ of fork-join processes with ℓ -nested fork nodes (i.e. at most 2^ℓ processes can be run in parallel) which modelizes Multi-BSP architectures with ℓ levels of components. The specification of such process is built the same way than a specification for simple varieties of trees of height ℓ :

$$\begin{aligned} \mathcal{F}_0 &= \text{SEQ}_{\geq 1} \mathcal{Z} \\ \mathcal{F}_\ell &= \mathcal{Z} + \mathcal{Z} \times \mathcal{F}_\ell + \mathcal{Z} \times \mathcal{F}_{\ell-1}^2 \times \mathcal{F}_\ell \end{aligned}$$

Thanks to the ordered product we can define a specification \mathcal{N}_ℓ for these fork-join processes with increasing labelings corresponding to their runs:

$$\begin{aligned} \mathcal{N}_0 &= \text{SET}_{\geq 1} \mathcal{Z} \\ \mathcal{N}_\ell &= \mathcal{Z} + \mathcal{Z}^\square * \mathcal{N}_\ell + \mathcal{Z}^\square * (\mathcal{N}_{\ell-1}^2 \boxtimes \mathcal{N}_\ell) \end{aligned}$$

Proposition 5.2. *The generating function N_ℓ of the class \mathcal{N}_ℓ verifies the following equations:*

$$\begin{cases} \mathcal{L}_c(N_0(z)) = \frac{z}{1-z} \\ \mathcal{L}_c(N_\ell(z)) = \frac{z}{1-z-z \mathcal{L}_c(\mathcal{N}_{\ell-1}) \odot \mathcal{L}_c(\mathcal{N}_{\ell-1})} \end{cases}$$

where $A(z) \odot B(z)$ is the colored product define in [BDGP17b] by $\mathcal{L}_c(\mathcal{B}_c(A(z)) \cdot \mathcal{B}_c(B(z)))$.

Proof: The derivation is direct using the following standard properties of the combinatorial Laplace and Borel transforms:

$$\mathcal{L}_c \left(\int A(z) \right) = z \mathcal{L}_c(A(z)) \quad \text{and} \quad \mathcal{L}_c(A(z)^2) = \mathcal{L}_c(A(z)) \odot \mathcal{L}_c(A(z))$$

□

Theorem 5.3. $\mathcal{L}_c(N_\ell)$ is a rational function with numerator $P_\ell(z)$ and denominator $Q_\ell(z)$ of degree d_ℓ such that:

$$\begin{cases} d_0 = 1 \\ d_\ell = \frac{(d_{\ell-1} + 1)(d_{\ell-1} + 2)}{2} \end{cases}$$

Moreover P_ℓ and Q_ℓ are coprimes and have only simple roots.

Proof: Before proving that claim by induction, we recall a basic property of combinatorial Laplace transform:

$$\mathcal{L}_c(e^{az}) = \frac{1}{1 - az}.$$

For the base case $N_0(z)$ the proof is direct: $N_0(z) = \exp(z) - 1$ and so $\mathcal{L}_c(N_0) = \frac{z}{1-z}$.

Now suppose, for some $\ell \geq 1$, that $N_{\ell-1}(z) = \frac{P_{\ell-1}(z)}{Q_{\ell-1}(z)}$ where $P_{\ell-1}$ and $Q_{\ell-1}$ are polynomials of degree $d_{\ell-1}$. Then by proposition 5.2 and induction hypothesis we have:

$$\mathcal{L}_c(N_\ell(z)) = \frac{z}{1 - z - z \left(\frac{P_{\ell-1}(z)}{Q_{\ell-1}(z)} \odot \frac{P_{\ell-1}(z)}{Q_{\ell-1}(z)} \right)}$$

By partial fraction decomposition we can write

$$\frac{P_{\ell-1}(z)}{Q_{\ell-1}(z)} = \gamma_{\ell-1} + \sum_{i=1}^{d_{\ell-1}} \frac{\alpha_i^{(\ell-1)}}{1 - \beta_i^{(\ell-1)} z},$$

where the α, β and γ are complex constants. So the combinatorial Borel transform of that function is a sum of $\alpha_i^{(\ell-1)} \exp(\beta_i^{(\ell-1)} z)$. Thus we have:

$$\begin{aligned} \frac{P_{\ell-1}(z)}{Q_{\ell-1}(z)} \odot \frac{P_{\ell-1}(z)}{Q_{\ell-1}(z)} &= \mathcal{L}_c \left(\gamma_{\ell-1}^2 + 2\gamma_{\ell-1} \sum_i \alpha_i^{(\ell-1)} \exp(\beta_i^{(\ell-1)} z) \right. \\ &\quad \left. + \sum_{i,j} \alpha_i^{(\ell-1)} \alpha_j^{(\ell-1)} \exp((\beta_i^{(\ell-1)} + \beta_j^{(\ell-1)}) z) \right) \end{aligned}$$

By Laplace transform, that sum of exponential factors becomes a partial fraction expansion containing $\frac{(d_{\ell-1}+1)(d_{\ell-1}+2)}{2}$ poles (the β_i and their product). All this poles are simples by induction hypothesis (all the β_i are different). Thus $\mathcal{L}_c(N_\ell(z))$ is a rational function with the claimed properties. \square

Because these are rational functions, indeed they are D-finite. Using the `ore_algebra` package for the Sagemath software (see [KJJ13] and [The18]) we were able to compute the differential equation for N_ℓ up to $\ell = 2$.

$$\mathcal{L}_c(N_2(z)) = \frac{-\frac{4}{5}x^{10} + \frac{51}{20}x^9 - \frac{639}{160}x^8 + \frac{2501}{640}x^7 - \frac{1627}{640}x^6 + \frac{2897}{2560}x^5 - \frac{11}{32}x^4 + \frac{87}{1280}x^3 - \frac{1}{128}x^2 + \frac{1}{2560}x}{x^{10} - \frac{333}{80}x^9 + \frac{77}{10}x^8 - \frac{1121}{128}x^7 + \frac{8729}{1280}x^6 - \frac{9647}{2560}x^5 + \frac{3811}{2560}x^4 - \frac{33}{80}x^3 + \frac{97}{1280}x^2 - \frac{21}{2560}x + \frac{1}{2560}}$$

The next ratio, for $\ell = 3$ is of degree 66, thus the calculation becomes very hard.

Corollary 4. *The number of runs of size n fork-join processes with a fork-depth of 2 verifies:*

$$\lim_{n \rightarrow \infty} [z^n] \mathcal{L}_c(N_2(z)) = \sigma \cdot \rho_2^{-n},$$

where $\sigma \cong 6.974 \cdot 10^{-5}$ and $\rho_2 \cong 1.852 \cdot 10^{-1}$ are both solutions of degree 10 polynomials.

The proof is a direct application of singularity analysis.

5.1.5 Hook-length formula

To conclude that section we present the hook-length formula (we introduced in [BDGP17b]). That formula has the benefit of emphasizing the correspondence between these 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 thanks to the following that formula.

For this, we need to define two kind of sub-structures found in their covering DAGs. Let P be a fork-join process. A *largest series component* X of P is a connected sub-process of P such that its direct ancestor is a fork node, and its direct descendant is the corresponding join node. The set of largest series components of P is denoted by $\mathcal{S}e_P$. Complementary, a *largest parallel component* Y of P is a disconnected sub-process composed by the two largest series components associated to the same pair of fork/join nodes. The set of largest parallel components of P is denoted by $\mathcal{P}a_P$.

Theorem 5.4. (Hook-length formula for fork-join processes). *The number of linear extensions of a fork-join process P is*

$$|\mathcal{L}e(P)| = \frac{\prod_{Y \in \mathcal{P}a_P} |Y|!}{\prod_{X \in \mathcal{S}e_P} |X|!}.$$

An application of the formula for our example in Fig. 6 gives $(2! 6! 2!) / (1! 1! 4! 2! 1! 1!) = 2880/48 = 60$. Thus there are 60 different linear extensions induced by our example.

Proof: Here we provide a new proof based on the BIT rules. The theorem can be demonstrated using Möhring's formula [M87], however a direct proof based on the integral formula of the BI-decomposition is proposed here.

The proof relies on an induction on the size of the process P . Suppose the result is correct for fork-join processes of size smaller than n . Take into account the process P of size n . First suppose P is a series of its root p and a second fork-join process Q . Thus the size of Q is $n - 1$. We apply the inductive assumption on Q .

$$|\mathcal{L}e(Q)| = \frac{\prod_{Y \in \mathcal{P}a_Q} |Y|!}{\prod_{X \in \mathcal{S}e_Q} |X|!}.$$

However, the last integration for Q in the context of P is between α and 1 instead of 0 and 1. Thus

$$\begin{aligned} \frac{|\mathcal{L}e(P)|}{|P|!} &= \int_0^1 \left(\int_0^\alpha \frac{|\mathcal{L}e(Q)|}{(|Q| - 1)!} (1 - q)^{|Q| - 1} dq \right) dp \\ &= \int_0^1 \frac{|\mathcal{L}e(Q)|}{|Q|!} (1 - p)^{|Q|} dp = \frac{|\mathcal{L}e(Q)|}{(|Q| + 1)!}. \end{aligned}$$

We deduce $|\mathcal{L}^{\mathcal{E}}(P)| = |\mathcal{L}^{\mathcal{E}}(Q)|$; furthermore $\mathcal{S}e_Q = \mathcal{S}e_P$ and $\mathcal{P}a_Q = \mathcal{P}a_P$, so the hook-length formula for P is satisfied.

Let us suppose P has a root p that is a fork node. We use its encoding as a tree to easily describe P . The root p has three subtrees P_1, P_2 and Q . The recursive strategy and the inductive assumption reduces all these three substructures to three nodes p_1, p_2 and q . The last integration for P_1 and P_2 are between p and q , thus

$$\Psi_{P_1} = \frac{|\mathcal{L}^{\mathcal{E}}(P_1)|}{(|P_1| - 1)!} (p_1 - q)^{|P_1| - 1}, \quad \Psi_{P_2} = \frac{|\mathcal{L}^{\mathcal{E}}(P_2)|}{(|P_2| - 1)!} (p_2 - q)^{|P_2| - 1}.$$

The last integration for Q is between p and 1, thus

$$\Psi_Q = \frac{|\mathcal{L}^{\mathcal{E}}(Q)|}{(|Q| - 1)!} (1 - q)^{|Q| - 1}.$$

Then we can, for example, reduce p_1, p_2, q and finally p with respectively the rules I, I, B and B. Thus

$$\frac{|\mathcal{L}^{\mathcal{E}}(P)|}{|P|!} = \int_0^1 \left(\int_p^1 \left(\int_p^q \left(\int_p^q \Psi_{P_1} dp_1 \right) \cdot \Psi_{P_2} dp_2 \right) \cdot \Psi_Q dq \right) dp.$$

Let us recall the following equation, proved by repeated integration by parts

$$\int_a^1 (1 - x)^r \cdot (x - a)^s dx = \frac{r! s!}{(r + s + 1)!} (1 - a)^{r+s+1}.$$

Using this last result we compute

$$\begin{aligned} \frac{|\mathcal{L}^{\mathcal{E}}(P)|}{|P|!} &= \int_0^1 \frac{(|P_1| + |P_2|)! \cdot |\mathcal{L}^{\mathcal{E}}(P_1)| \cdot |\mathcal{L}^{\mathcal{E}}(P_2)| \cdot |\mathcal{L}^{\mathcal{E}}(Q)|}{|P_1|! \cdot |P_2|! \cdot (|P_1| + |P_2| + |Q|)!} (1 - p)^{|P_1| + |P_2| + |Q|} dp \\ &= \binom{|P_1| + |P_2|}{|P_1|} \frac{|\mathcal{L}^{\mathcal{E}}(P_1)| \cdot |\mathcal{L}^{\mathcal{E}}(P_2)| \cdot |\mathcal{L}^{\mathcal{E}}(Q)|}{(|P_1| + |P_2| + |Q| + 1)!}. \end{aligned}$$

Furthermore, $\mathcal{S}e_P = \mathcal{S}e_{P_1} \cup \mathcal{S}e_{P_2} \cup \mathcal{S}e_Q \cup \{P_1, P_2\}$ and $\mathcal{P}a_P = \mathcal{S}e_{P_1} \cup \mathcal{S}e_{P_2} \cup \mathcal{S}e_Q \cup \{(P_1, P_2)\}$. \square

Corollary 5. *For a fork join process of size n the counting problem is of time complexity $O(n)$ in number of arithmetic operations.*

It exists a uniform sampler using an optimal number of random bits (up to a constant factor) with time complexity $O(n\sqrt{n})$ on average.

Proof sketch: The counting algorithm is easily derived from the hook-length formula. First we need to compute and memoize the values of the factorial of the integers from 1 to n . Then a traversal of the graph in a “bottom-up” fashion allows to collect the sizes of the largest series and parallel components. At each step (a linear number) a constant number of arithmetic operations is done (because the precomputation of the factorials), and so the $\mathcal{O}(n)$ complexity.

The uniform sampler proceeds by induction. If the process falls in the $\mathcal{Z} \times \mathcal{F}$ class then it draws a linear extension of the sub-process in \mathcal{F} prefixed by an action. Else, the process falls in the $\mathcal{Z} \times \mathcal{F}^2 \times \mathcal{F}$ class. In that case a linear extension is sampled for each subprocess, then the two extensions of the up

processes are shuffled and concatenated to the one of the bottom process. The number of random bit used by the shuffling procedure is the key to achieve the optimality claimed. Details are given in [BDGP17a] and out of the scope here. To show the $O(n\sqrt{n})$ time complexity note that each vertex is manipulated a number of times proportional to its depth in the tree-like structure, and so the sum of these numbers is proportional to the path length of the tree: $O(n\sqrt{n})$ in average (see [FS09]). \square

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 $Q \equiv \langle B \rangle.0$

Fig. 7: A proof system for promises.

In Fig. 7 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 [BDGV18]. An arch-process follows the constraint of Fig. 7 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 [BDGV18] 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.

In Fig. 8 (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 5.5. *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.*

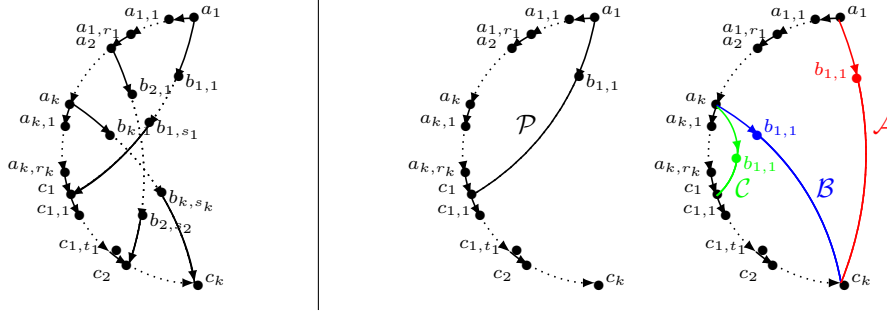


Fig. 8: The structure of an arch-process (left) and the inclusion-exclusion counting principle (right).

Proof: Start with a promise-process \mathcal{P} and denote by $\ell_{\mathcal{P}}$ its number of executions. Suppose, in our promise-process \mathcal{P} we have $s_1 > 1$. Let us first derive a process starting in action a_1 containing at least one promise (also starting at a_1) built on a single action $b_{1,1}$, instead of a sequence of actions $(b_{1,s})_{s=1,\dots,s_1}$ with $s_i > 1$. We can replace in \mathcal{P} the single first promise by two distinct promises both starting at a_1 and ending at c_1 , the first promise containing only the action $b_{1,1}$ and the second one containing the rest of the sequence $b_{1,2}, \dots, b_{1,s_1}$: this new process is denoted by $\tilde{\mathcal{P}}$. This transformation increases the number of executions of the DAG. In fact, now $b_{1,1}$ is no more enforced to appear before $b_{1,2}$. Thus it could appear everywhere in between of $b_{1,i}$ and $b_{1,i+1}$ for all $i \in \{2, \dots, s_1 - 1\}$ or also after b_{1,s_1} . Thus we get $\ell_{\mathcal{P}} = \ell_{\tilde{\mathcal{P}}} / s_1$.

Remark, now we have a process $\tilde{\mathcal{P}}$ such that by removing its node $b_{1,1}$ is still a promise process. Let us now introduce some inclusion-exclusion argument in order to count the number of executions of $\tilde{\mathcal{P}}$. Let us start by the key-idea, but as we will see it must be refined in order to get a well defined sub-problem. If we replace the synchronization of $b_{1,1}$ in c_1 , later in the control thread by another in c_k , then we allow new executions that are not correct for $\tilde{\mathcal{P}}$ thus in order to remove them we remove the number of executions of the process where a new promise starting at c_1 and synchronizing at c_k and containing only $b_{1,1}$.

In Fig. 8 we go one step further. There we focus on the control thread and the promise associated to $b_{1,1}$. To obtain a clear representation we omit to draw the other promises. Thus the representation associated to $\tilde{\mathcal{P}}$ is the leftmost one, in black. Let us denote the partially colored processes \mathcal{A} (in red), \mathcal{B} (in blue) and \mathcal{C} (in green). Thus the number of executions $\ell_{\tilde{\mathcal{P}}} = \ell_{\mathcal{A}} - \ell_{\mathcal{B}} + \ell_{\mathcal{C}}$.

Let us denote by $\tilde{\mathcal{A}}$ the process \mathcal{A} where $b_{1,1}$ is removed. The executions of $\tilde{\mathcal{A}}$ are such that $b_{1,1}$ can appear everywhere between a_1 and c_k in the executions of $\tilde{\mathcal{A}}$. Thus $\ell_{\tilde{\mathcal{A}}} = (n - 2) \cdot \ell_{\tilde{\mathcal{A}}}$. And remark that $\tilde{\mathcal{A}}$ is a promise process (of size $n - 1$), thus we can go recursively inside it to compute its number of executions.

In the process \mathcal{B} , we can insert the action $b_{1,1}$ in the front of the promise starting at a_k , i.e. just before $b_{k,1}$. Doing this reduces the numbers of executions by a factor $1/(s_k + 1)$ and the new process is now a promise process. Repeating the process, we will manage to remove entirely the first promise, and thus, even if we do not manage to reduce the number of nodes of the process, we remove one of its promise, thus we can continue recursively and obtain an extreme case with a single promise (and its number of executions is easy to calculate).

Finally, for process \mathcal{C} we can insert $b_{1,1}$ just before $a_{k,1}$ but this choice reduces the numbers of executions by a factor $1/(r_k + 1)$ but the new process is now a promise process. With the same argument as

before we can continue recursively.

Finally, the proof of Theorem 5.5, for a promise process \mathcal{P} is derived from the fact that you must consider all promise processes induced by \mathcal{P} where the only two sequences that can change are $(a_{k,r})_{r=1,\dots,r_k}$ and $(b_{1,s})_{s=1,\dots,s_k}$: both can be increased at most by n nodes. Thus we deduce that using a dynamical programming approach with memoization of the calculated values gives the value $\ell_{\mathcal{P}}$ is $O(n^2)$ arithmetic operations. \square

From this counting procedure we developed an uniform random sampler following the principles of the *recursive method*, as described in [FZC94].

Algorithm 2 Uniform random sampling

We suppose here that all the promises do contain a single action. We must take care of a factor in the counting part of the algorithm.

```

1: function SAMPLING( $\mathcal{A}$ )
2:   if PromiseCount( $\mathcal{A}$ ) = 0 then
3:     return ControlThread( $\mathcal{A}$ )
4:    $r :=$  RAND_INT( $1, \ell_{\mathcal{A}}$ )
5:    $pos := 1 +$  StartPosition( $\mathcal{A}, 1$ )
6:    $\bar{\mathcal{A}} :=$  RemovePromise( $\mathcal{A}, 1$ )
7:   while  $r > 0$  and  $pos \leq$  EndPosition( $\mathcal{A}, 1$ ) do
8:      $\bar{\mathcal{A}} :=$  InsertControlThread( $\bar{\mathcal{A}}, pos, b_{1,1}$ )
9:      $r := r - \ell_{\bar{\mathcal{A}}}$ 
10:     $pos := pos + 1$ 
11:  return SAMPLING( $\bar{\mathcal{A}}$ )

```

The function PromiseCount(\mathcal{A}) returns the numbers of promises of the process \mathcal{A} .

The function ControlThread(\mathcal{A}) returns the sequence of actions in the main control thread of \mathcal{A} .

The function RAND_INT(a, b) returns uniformly sampled integer between a and b included.

The function StartPosition($\mathcal{A}, 1$) returns the position of the postpone action related to the first promise.

The function RemovePromise($\mathcal{A}, 1$) removes the first promise of first promise \mathcal{A} .

The function EndPosition($\mathcal{A}, 1$) returns the position of the synchronization action related to the first promise.

The function InsertControlThread($\bar{\mathcal{A}}, pos, b_{1,1}$) inserts the action associated to the first promise $b_{1,1}$ in the control thread of $\bar{\mathcal{A}}$, at position pos returns the position of the synchronization action related to the first promise.

Theorem 5.6. *Let \mathcal{P} be a promise-process of size n with $k \geq n$ promises. Algorithm 2 is a uniform sampler of the linear extensions of \mathcal{P} with $O(n^4)$ time-complexity in the number of arithmetic operations.*

Here we remark a big combinatorial change by comparing promise processes to arch-processes (from paper [BDGV18]). In fact, in the later case the sub-problem induced by the second process (associated to \mathcal{B}) was exactly the same as the one of \mathcal{P} . And thus, there the uniform recursive sampling could be obtained efficiently in $O(k)$ arithmetic operations (once a quadratic time complexity pre-computation has been memoized). Here the pre-computation is harder.

Proof idea: 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 $O(n^2)$ (in [BDGV18]) to $O(n^4)$ here. \square

6 Experimental study

Algorithm	Class	Count.	Unif. Rand. Gen.	Reference
FJ	Fork-join	$O(n)$	$O(n \cdot \sqrt{n})$ on average	[BDGP17a]
ARCH	Arch-processes	$O(n^2)$	$O(n^4)$ worst case	[BDGV18]/Theorem 5.6
BIT	BIT-decomposable	?	?	Theorem 3.1
CFTP ^(vi)	All processes	–	$O(n^3 \cdot \log n)$ expected	[Hub06]

Tab. 1: 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 algorithms for counting and generating process executions uniformly at random. Tab. 1 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.

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^(vii) 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 [BDGP17a] 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

^(vi) 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.

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

^(viii) Arbogen is uniform random generation for context-free grammar structures: cf. <https://github.com/fredokun/arbogen>.

FJ size	$\#\mathcal{LE}$	FJ gen	(count)	BIT gen	(count)	CFTP gen
10	19	0.00001 s	(0.0002 s)	0.00006 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	$\#\mathcal{LE}$	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

Tab. 2: Benchmark results for BIT-decomposable classes: FJ and Arch.

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 pre-computation. As an illustration, for arch-processes of size 100 with 32 arches, the sampler becomes about 500 times faster. However the memory requirement for the pre-computation 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 [BGP13]). 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 [BDGV18])

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.

References

- [AB09] S. Arora and B. Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.
- [AM15] S. Abbes and J. Mairesse. Uniform generation in trace monoids. In *MFCS 2015*, volume 9234 of *LNCS*, pages 63–75. Springer, 2015.
- [BDGP17a] O. Bodini, M. Dien, A. Genitrini, and F. Peschanski. Entropic Uniform Sampling of Linear Extensions in Series-Parallel Posets. In *12th International Computer Science Symposium in Russia (CSR)*, pages 71–84, 2017.
- [BDGP17b] O. Bodini, M. Dien, A. Genitrini, and F. Peschanski. The Ordered and Colored Products in Analytic Combinatorics: Application to the Quantitative Study of Synchronizations in Concurrent Processes. In *14th SIAM Meeting on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 16–30, 2017.
- [BDGP19] O. Bodini, M. Dien, A. Genitrini, and F. Peschanski. The combinatorics of barrier synchronization. In *Proceedings of the 40th International Conference on Application and Theory of Petri Nets and Concurrency, PETRI NETS 2019*, pages 386–405, 2019.
- [BDGV18] O. Bodini, M. Dien, A. Genitrini, and A. Viola. Beyond series-parallel concurrent systems: The case of arch processes. In *Analysis of Algorithms, AofA 2018*, volume 110 of *LIPICs*, pages 14:1–14:14, 2018.
- [BGP13] O. Bodini, A. Genitrini, and F. Peschanski. The combinatorics of non-determinism. In *FSTTCS’13*, volume 24 of *LIPICs*, pages 425–436. Schloss Dagstuhl, 2013.
- [BGP16] O. Bodini, A. Genitrini, and F. Peschanski. A Quantitative Study of Pure Parallel Processes. *Electronic Journal of Combinatorics*, 23(1):P1.11, 39 pages, 2016.
- [BLL98] F. Bergeron, G. Labelle, and P. Leroux. *Combinatorial Species and Tree-like Structures*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1998.
- [BMS17] N. Basset, J. Mairesse, and M. Soria. Uniform sampling for networks of automata. In *Concur 2017*, volume 85 of *LIPICs*, pages 36:1–36:16. Schloss Dagstuhl, 2017.

- [BMW18] C. Banderier, P. Marchal, and M. Wallner. Rectangular Young tableaux with local decreases and the density method for uniform random generation (short version). In *GASCom 2018*, Athens, Greece, June 2018.
- [BW91] G. Brightwell and P. Winkler. Counting linear extensions is #P-complete. In *STOC*, pages 175–181, 1991.
- [FS09] P. Flajolet and R. Sedgewick. *Analytic Combinatorics*. Cambridge Univ. Press, 2009.
- [FZC94] P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theor. Comput. Sci.*, 132(2):1–35, 1994.
- [GS05] R. Grosu and S. A. Smolka. Monte carlo model checking. In *TACAS'05*, volume 3440 of *LNCS*, pages 271–286. Springer, 2005.
- [GV94] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *J. Parallel Distrib. Comput.*, 22(2):251–267, 1994.
- [HFM88] D. Hensgen, R. A. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.
- [Hub06] M. Huber. Fast perfect sampling from linear extensions. *Discrete Mathematics*, 306(4):420–428, 2006.
- [KJJ13] M. Kauers, M. Jaroschek, and F. Johansson. Ore polynomials in sage. *CoRR*, abs/1306.4263, 2013.
- [LS88] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI'88*, pages 260–267. ACM, 1988.
- [M87] R. H. Möhring. *Computationally Tractable Classes of Ordered Sets*. Institut für Ökonometrie und Operations Research: Report. 1987.
- [ODG⁺11] J. Oudinet, A. Denise, M.-C. Gaudel, R. Lassaigne, and S. Peyronnet. Uniform monte-carlo model checking. In *FASE 2011*, volume 6603 of *LNCS*. Springer, 2011.
- [Riv88] I. Rival, editor. *Algorithms and Order*. NATO Science Series. Springer, 1988.
- [Sen07] K. Sen. Effective random testing of concurrent programs. In *Automated Software Engineering ASE'07*, pages 323–332. ACM, 2007.
- [Sta86] R. P. Stanley. Two poset polytopes. *Discrete & Computational Geometry*, 1:9–23, 1986.
- [The18] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.5.0)*, 2018. <https://www.sagemath.org>.
- [Val11] L. G Valiant. A bridging model for multi-core computing. *Journal of Computer and System Sciences*, 77(1):154–166, 2011.

A Appendix: Extended semantics

In this appendix we give a detailed proof for Theorem 2.1, which establishes the connection between processes and their control graph. One limitation of the semantics given in the main body of the paper is that deadlocks are not recorded : deadlocked executions simply stops.

$$\begin{array}{c}
\frac{}{\alpha.P \xrightarrow{\alpha} P} \text{ (eact)} \quad \frac{}{\langle B \rangle P \xrightarrow{\langle B \rangle} P} \text{ (esig)} \quad \frac{P \xrightarrow{\langle B \rangle} P' \quad Q \xrightarrow{\langle B \rangle} Q'}{P \parallel Q \xrightarrow{\langle B \rangle} P' \parallel Q'} \text{ (ejoin)} \\
\\
\frac{P \xrightarrow{\mu} P'}{P \parallel Q \xrightarrow{\mu} P' \parallel Q} \text{ (elpar)} \quad \frac{Q \xrightarrow{\mu} Q'}{P \parallel Q \xrightarrow{\mu} P \parallel Q'} \text{ (erpar)} \\
\\
\frac{P \xrightarrow{\mu} P' \quad \mu \neq \overline{\langle B \rangle}}{\nu(B) P \xrightarrow{\mu} \nu(B) P'} \text{ (elift)} \quad \frac{P \xrightarrow{\langle B \rangle} P'}{\nu(B) P \xrightarrow{\langle B \rangle} P'} \text{ (esync)}
\end{array}$$

Fig. 9: Variant of the semantics with explicit barriers.

We thus consider in Fig. 9 a more detailed semantics that preserve all the information of the process executions, especially by keeping track of the barrier used in the synchronization steps.

Proposition A.1. $P \xrightarrow{\alpha} P' \implies \exists B_1, \dots, B_n (n \geq 0), P \xrightarrow{\langle B_1 \rangle} \dots \xrightarrow{\langle B_n \rangle} P_\alpha \xrightarrow{\alpha} P'$.

Proof: This is by rule induction on the standard semantics. \square

This means that any execution σ of the standard semantics can be translated to an extended execution $\bar{\sigma}$ with explicit barriers.

Definition A.1 (Extended execution of a process). An *extended execution* $\bar{\sigma}$ of P is a finite sequence $\langle \mu_1, \dots, \mu_n \rangle$ such that there is a set of processes $P'_{\mu_1}, \dots, P'_{\mu_n}$ and a path $P \xrightarrow{\mu_1} P'_{\mu_1} \dots \xrightarrow{\mu_n} P'_{\mu_n}$ with $P'_{\mu_n} \Rightarrow$. The extended behavior of a process P is the set of all its extended executions.

An important property is that even for a deadlocked process there exists (at least) an extended execution eventually reaching a termination.

Proposition A.2. *If P is not a termination (e.g. $0, 0 \parallel 0$, etc.), then $\exists \mu, P', P \xrightarrow{\mu} P'$.*

Proof: This is trivial by induction on the syntax since except for terminated processes (e.g. 0 or an equivalent form such as $(\nu B)0, 0 \parallel 0$, etc.) it is a simple fact that at least one rule of Fig. 9 is enabled. \square

Now the connection between normal and extended executions is straightforward.

Proposition A.3. *Let P a deadlock-free process and $\bar{\sigma}$ one of its extended executions. Then there is a normal execution σ of P that is exactly $\bar{\sigma}$ with all its explicit barriers removed.*

Proof: This is by definition of the executions and Proposition A.1, of course assuming that deadlock-free process always have normal transition until their completion. \square

We now promote the causal relations to extended executions.

Definition A.2 (Extended cause, extended direct cause). Let P be a process. An action α of P is said an *extended cause* of another action β , denoted by $\alpha \bar{\leq} \beta$, iff for any extended execution $\bar{\sigma}$ of P we have $\bar{\sigma}(\alpha) \leq \bar{\sigma}(\beta)$. Moreover, α is an *extended direct cause* of β , denoted by $\alpha \bar{<} \beta$ iff $\alpha \bar{\leq} \beta$ and there is no γ such that $\alpha \bar{<} \gamma \bar{<} \beta$.

For deadlock-free processes the normal and extended causal relation coincide.

Proposition A.4. *Let P a deadlock-free process. Then $\alpha \bar{<} \beta$ iff $\alpha < \beta$.*

Proof: This is a direct consequence of Proposition A.3. \square

We are now concerned with *deadlocked* processes.

Proposition A.5. *A process P has a deadlock if and only if there is an extended execution $\bar{\sigma}$ and a barrier B such that the event $\langle B \rangle$ is repeated at least twice in $\bar{\sigma}$.*

Proof: A simple observation is that the only rule that can generate an immediate deadlock is (sync). So a deadlocked process P must have a subprocess of the form $\nu(B) Q$ such that rule (sync) only can be triggered but for $\text{sync}_B(Q) = Q'$ we have $\text{wait}_B(Q') = \text{true}$. In the extended executions the event $\langle B \rangle$ will still be recorded for Q . But going back to the standard semantics, there must be one of the subprocesses of Q' of the form $\langle B \rangle R$ since $\text{wait}_B(Q') = \text{true}$ and such that Q' is distinct from Q (otherwise the deadlock is caused by another barrier). Eventually in at least one of the executions of Q' another event $\langle B \rangle$ will occur because the extended executions are guaranteed deadlock-free (by Proposition A.2). Finally, since Q' is a derivative of Q it must be the case that the event $\langle B \rangle$ occurs twice in at least one execution σ going through both Q and Q' . \square

Hitherto, we have all the required properties concerning the extended executions, we thus turn to the control graph construction, now extended with explicit barriers.

Definition A.3 (Construction of extended control graphs). Let P be a process term. Its extended control graph is $\text{ectg}(P) = \langle V, E \rangle$, constructed inductively as follows:

$$\left[\begin{array}{l} \text{ectg}(0) = \langle \emptyset, \emptyset \rangle \\ \text{ectg}(\alpha.P) = \alpha \rightsquigarrow \text{ectg}(P) \\ \text{ectg}(\nu(B)P) = \text{ectg}(P) \\ \text{ectg}(\langle B \rangle P) = \langle B \rangle \rightsquigarrow \text{ectg}(P) \\ \text{ectg}(P \parallel Q) = \text{ectg}(P) \cup \text{ectg}(Q) \end{array} \right.$$

The main difference with the normal control graph is that the barrier synchronizations are not removed along the construction.

If we only consider the atomic actions, then we have the very interesting property that the normal and extended control graph indeed coincide. We denote by $\alpha \rightsquigarrow^+ \beta$ a path in $\text{ectg}(P)$ such that α and β are atomic actions, and in the considered path only barrier events may occur.

Proposition A.6. *$\alpha \rightsquigarrow^+ \beta \in \text{ectg}(P)$ if and only if $\alpha \rightsquigarrow \beta \in \text{ctg}(P)$.*

Proof: This is trivial given the similarity of the definitions of ctg and ectg . As long as only the atomic actions (and not the barrier events) are considered, the definition generate exactly the same dependencies, although it might be the case that many barrier events must be traversed from α in order to reach β . \square

Moreover, there is now a bijection between the extended control graph edges and the extended direct causes.

Proposition A.7. $\alpha \rightsquigarrow^+ \beta \in \text{ectg}(P)$ iff $\alpha \prec \beta$.

Proof: This derives easily from the propositions above. \square

We now have all the building blocks for our main proof.

Proof of Theorem 2.1: If P is deadlocked then we know a barrier event $\langle B \rangle$ occurs at least twice in a given extended execution $\bar{\sigma}$ (according to Proposition A.5). Moreover, the two occurrences cannot be consecutive otherwise the rule (ejoin) would have collapsed them initially. Hence there is at least an action that is at the same time a cause for and caused by the event $\langle B \rangle$! Put in other terms we have a cycle in both $\text{ectg}(P)$ and $\text{ctg}(P)$. Now if we consider a deadlock-free process P then if $\alpha \rightsquigarrow \beta \in \text{ctg}(P)$ we have $\alpha \rightsquigarrow^+ \beta \in \text{ectg}(P)$ (by Proposition A.6) hence $\alpha \prec \beta$ (by Proposition A.7). Finally, by Proposition A.4 we can conclude the proof. \square

B Appendix: ordered and colored products

B.1 Reminders on Borel and Laplace transforms

Let us recall here classical relations between combinatorial Laplace transform and the traditional Laplace transform. By definition, the traditional Laplace transform is defined by $\mathcal{L}f = \int_0^\infty \exp(-zt)f(t)dt$ instead of $\mathcal{L}_c f = \int_0^\infty \exp(-t)f(zt)dt$.

This operator is clearly linear. By a simple change of variable, we get that $\mathcal{L}f(z) = \frac{1}{z} (\mathcal{L}_c f) (\frac{1}{z})$ or equivalently $\mathcal{L}_c f(z) = \frac{1}{z} (\mathcal{L}f) (\frac{1}{z})$ (Notice the perfect involution !)

Laplace transforms admit a functional inverse called Borel transforms. This transform also has an integral representation: for traditional Laplace transforms, the Borel transform is

$$\mathcal{B}(f) = \frac{1}{2i\pi} \int_{c-i\infty}^{c+i\infty} \exp(zt)f(t)dt$$

where c is greater than the real part of all singularities of $f(t)$.

By analogy, the combinatorial Borel transform is $\mathcal{B}_c(f) = \frac{1}{2i\pi} \int_{c-i\infty}^{c+i\infty} \frac{\exp(zt)}{t} f(1/t)dt$ where c is greater than the real part of all singularities of $f(1/t)/t$. The link with traditional Borel transforms is $\mathcal{B}_c(f) = \mathcal{B}(1/zf(1/z))$ or equivalently $\mathcal{B}(f) = \mathcal{B}_c(f(1/z))' = \mathcal{B}_c(1/zf(1/z))$

Now, let us essentially concentrate our attention on combinatorial transforms. Combinatorial Laplace transforms create a bridge between exponential generating functions ($\sum_{n \geq 0} a_n \frac{z^n}{n!}$) and ordinary generating functions ($\sum_{n \geq 0} a_n z^n$). Precisely, we have:

$$\mathcal{L}_c \left(\sum_{n \geq 0} a_n \frac{z^n}{n!} \right) = \sum_{n \geq 0} a_n z^n$$

Reciprocally, we have

$$\mathcal{B}_c\left(\sum_{n \geq 0} a_n z^n\right) = \sum_{n \geq 0} a_n \frac{z^n}{n!}$$

From those formulas on formal series, one can easily derive the following identities:

- $\mathcal{L}_c f' = \frac{1}{z}(\mathcal{L}_c f - f_0)$
- $\mathcal{L}_c(\int f) = z\mathcal{L}_c f$
- $\mathcal{B}_c(zf) = \int \mathcal{B}_c f$
- $\mathcal{B}_c\left(\frac{f-f_0}{z}\right) = (\mathcal{B}_c f)'$

As for traditional Laplace transforms, the product of Laplace transform can be express using convolution product. We have :

$$z\mathcal{L}_c f \times \mathcal{L}_c g = \mathcal{L}_c\left(\int_0^z f(t)g(z-t)dt\right).$$

Or equivalently,

$$\mathcal{L}_c f \times \mathcal{L}_c g = \mathcal{L}_c\left(\int_0^z f(t)g'(z-t)dt + g_0 f(z)\right).$$

Observe that the ordered product, in fact, gives a combinatorial interpretation of this adapted convolution. We denote by $f * g$ the combinatorial convolution $\int_0^z f(t)g'(z-t)dt + g_0 f(z)$.

The product of combinatorial Borel transforms can also be expressed with convolution in the complex plane as follow: using the traditional

$$\mathcal{B}f \times \mathcal{B}g = \mathcal{B}\left(\frac{1}{2i\pi} \int_{c-i\infty}^{c+i\infty} f(t)g(z-t)dt\right),$$

and compose it with the latter identities leads to the following formula

$$\mathcal{B}_c f \times \mathcal{B}_c g = \mathcal{B}_c\left(\frac{1}{2i\pi} \int_{c-i\infty}^{c+i\infty} \frac{1}{(1-zt)t} f(1/t)g(z/(1-zt))dt\right).$$

Proof of Proposition 5.1: Using Definition 5.1, we note that an object from \mathcal{C} is given by an object from \mathcal{A} and one from \mathcal{B} only by shifting the labels of the second one. Thus the number of objects of size n in \mathcal{C} is given by $\sum_{k=1}^{n-1} A_k \cdot B_{n-k}$. The result of the composition $\mathcal{B}(\mathcal{L}(A(z)) \cdot \mathcal{L}(B(z)))$ gives directly this sum. □