Associativity for Binary Parallel Processes: a Quantitative Study*

Olivier Bodini[†], Antoine Genitrini[‡§], Frédéric Peschanski[‡§] and Nicolas Rolin[†]
{Olivier.Bodini,Nicolas.Rolin}@lipn.univ-paris13.fr,
{Antoine.Genitrini,Frederic.Peschanski}@lip6.fr.

We investigate the common interpretation of parallel processes as computation trees. The basis for our approach is the combinatorics of increasingly labelled structures, and our main objective is to provide quantitative results relying on advanced analytic techniques. Unlike previous works, the combinatorial model we propose captures the following ingredients of the algebraic presentation : a binary parallel operator with associativity law. The switch from general trees to binary encodings in this paper makes everything more complex (but eventually workable). Ultimately, we provide a precise characterization and asymptotic approximations of various measures of parallel processes in the average case, especially the average size of the computation trees and their average number of paths, providing a more meaningful notion of *combinatorial explosion* than in the (rather trivial) worst-case. Beyond the measures, we also provide a precise characterization of the typical combinatorial shape of the computation trees, especially their level-decomposition, an interesting notion of process depth. From a more practical point of view, we develop efficient algorithms for the uniform random sampling of computations. Thanks to our typical shape analysis, it is possible to uniformly sample computation prefixes at a given depth in a very efficient way. Indeed, these algorithms work directly on the syntax trees of the processes and do not require the explicit construction of the state space, hence completely avoiding the combinatorial explosion.

*This research was partially supported by the A.N.R. project *MAGNUM*, ANR 2010-BLAN-0204.
[†]Laboratoire d'Informatique de Paris-Nord, CNRS UMR 7030 - Institut Galilée - Université Paris-Nord, 99, avenue Jean-Baptiste Clément, 93430 Villetaneuse, France.
[‡]Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France.
[§]CNRS, UMR 7606, LIP6, F-75005, Paris, France.

1 Introduction

The combinatorial study of concurrent processes is a relatively recent and quite active area of research. Pure notions of parallelism are studied from different perspectives in the literature. The *shuffle product on regular words* provides an automata-theoretic interpretation that received much attention (cf. e.g. [1, 17]). The *trace monoid* provides another mathematical characterization of pure parallelism, which was also extensively studied (cf. e.g. [15]). On the other hand, we investigate the more common and concrete interpretation of parallel processes as *computation trees*. Despite its straightforward algebraic characterization, the underlying structures based on *increasing labellings* are quite intricate and their study represents a real challenge in terms of analytic combinatorics.

In [4] we provide an interpretation of non-determinism in terms of labelled tree-shaped structures. In the paper [3] (and its extended version [5] currently under submission) we demonstrate a one-to-one correspondence between the sub-case of pure parallel processes and the well-known combinatorial class of *general increasing trees* [2]. This leads to many quantitative results, most notably the average number of concurrent runs for syntaxic process, i.e. an average-case analysis of the *combinatorial explosion effect*. We also develop an algorithmic framework for statistical model checking based on the uniform random generation of concurrent runs directly from the syntax, that is without having to construct explicitly the computation trees hence *avoiding* the combinatorial explosion.

However, an important simplification is imposed on the model in these preliminary works. The parallel operator is of an arbitrary arity, which allows us to consider *general trees* in the combinatorial interpretations. In most presentations of process algebras (cf. e.g. [16, 9]), on the contrary, a binary parallel operator is considered. It is well-known that general trees can be encoded in a binary form, hence the change is relatively transparent at the syntax level. However, as far as the semantic interpretation is concerned the correspondence between the two is highly non-trivial, as it is emphasized in this paper.

At the technical level, the main contributions of the paper are the following ones. In the theory, we provide a precise characterization and asymptotic approximations of various measures of parallel processes in the average case, especially: (1) the average number of computations (i.e. number of runs), and (2) the average size of computation trees. These provide a rather precise meaning of *combinatorial explosion*. Parallel processes are not just "exponential" in the worst case. Perhaps even more interestingly we characterize the typical combinatorial shape of the computation trees, based on a level-decomposition that proves somewhat unexpectedly *workable*. This provides an interesting interpretation for the notion of *process depth*.

From a more practical point of view, we show that the uniform random generation algorithm of [3, 5] can be adapted to the binary model. We also provide, thanks to our level-decomposition of computation trees, an extension of the algorithmic framework to allow efficient uniform samplings of computation prefixes. Hence if one may not explore a complete computation tree given its exponential size, we may exploit the fact that the tree prefixes grow rather "slowly" (although still in an exponential way). Developing these techniques for more expressive concurrent calculi (with e.g. non-deterministic choice as in [4]) would naturally lead to a mix of statistical and bounded exploration techniques for model-checking of large-scale concurrent systems. Indeed, these algorithms work directly on the syntax objects and do not require the explicit construction of the state space, hence avoiding the combinatorial explosion.

2 Context

2.1 Syntax: process trees

In this paper, pure parallel processes are specified using the following grammar:

- an atomic action, denoted $a, b, c, \ldots$ is a process,
- the prefixing $a.P$ of an action a and a process P is a process,
- the composition $P_1 \parallel P_2$ of exactly two processes P_1 and P_2, is a process.

The following process is the *running example* that will be used as illustration throughout the paper:

$$(a.b) \parallel [(c \parallel d) \parallel (e.(f \parallel g))].$$

Such process terms can be naturally interpreted as tree structures, as depicted on the left of Fig. 1. Pure parallel processes are composed out of unary action nodes, action leaves and binary parallel nodes. Process terms that are well-formed according to the grammar above will thus be called *process trees* from now on.

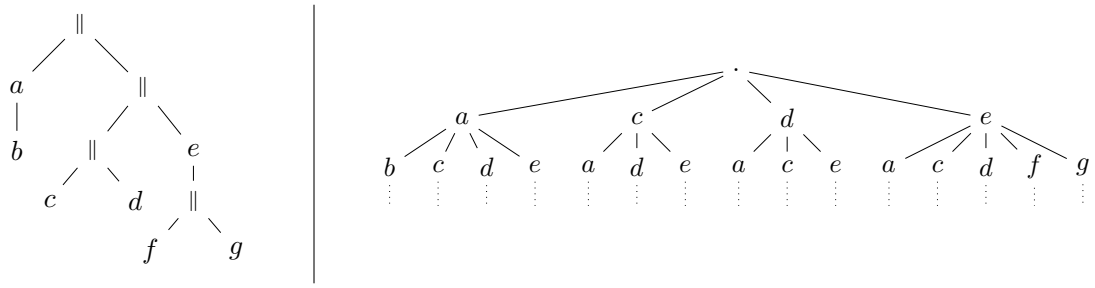

Figure 1: *A process tree of size* 7 *(left) and the first levels of its computation tree (right).*

A grammar for (finite) tree-shaped structures can be almost directly reinterpreted as a *combinatorial class*, only requiring a precise definition for the sizes of the objects belonging to the class. Because for the questions – about pure parallelism – that concern us the identity of the atomic actions will not play any role, our combinatorial specification will abstract from them. This leads to the following specification:

Definition 1. *The combinatorial class of process trees is specified as followed:*

$$\mathcal{P} = \mathcal{Z} + \mathcal{Z} \times \mathcal{P} + \mathcal{P} \times \mathcal{P}, \textit{where } \mathcal{Z} \textit{ marks the nodes containing an action.}$$

This can be read almost as a grammar: an object in class $\mathcal{P}$ is either a leaf (or external node) marked by $\mathcal{Z}$, a unary internal node also marked by $\mathcal{Z}$ and with a subtree in $\mathcal{P}$, or an unmarked binary nodes with two subtrees in $\mathcal{P}$. The marker $\mathcal{Z}$ explains which nodes must be counted in the resulting size of the object. Here we mark the nodes corresponding to actions in the grammar. Hence, the size of a process tree P is the number of occurrences of actions in the tree, and it is denoted by $|P|$. For example, the size of the running example is 7.

Let us now introduce some notations[1] about the class $\mathcal{P}$ of objects. Let us denote by P_n the number of processes of size n (also called the *counting sequence* of $\mathcal{P}$) and $P(z)$ the

[1]For all the combinatorial classes that will appear in the paper we will use the same kind of notations like $\mathcal{P}$ for the class, P_n for the counting sequence and $P(z)$ for the generating function.

ordinary generating function[2] related to the class $\mathcal{P}$: it satisfies $P(z) = \sum_{n>0} P_n\ z^n$. And the notation for the coefficient extraction of the generating function is $[z^n]P(z) = P_n$. Using Definition 1 and the *symbolic method* (cf.[12]), we deduce a functional equation from the specifications of processes: $P(z) = z + z \cdot P(z) + P(z) \cdot P(z)$. Here we give the first coefficients: $P(z) = z + 2\ z^2 + 6\ z^3 + 22\ z^4 + 90\ z^5 + 394\ z^6 + \ldots$. (e.g. there are 90 trees of size 5 in class $\mathcal{P}$).

Proposition 2. *The combinatorial class $\mathcal{P}$ satisfies:*

$$P(z) = \frac{1 - z - \sqrt{1 - 6z + z^2}}{2} \qquad \textit{and} \qquad P_n \sim_{n\to\infty} \sqrt{\frac{3\sqrt{2} - 4}{4\ \pi\ n^3}} \cdot \left(3 - 2\sqrt{2}\right)^{-n}.$$

This is a direct result of applying the symbolic method (cf. [12]), even if our way of counting nodes in not standard. We recall that $\rho_P = 3 - 2\sqrt{2}$ is the dominant singularity of P: it is directly associated to the exponential growth of $(P_n)_n$.

In [3, 5] we describe a variant of pure parallel processes in which the parallel operator is of an arbitrary arity. In terms of combinatorics, this is a much simpler setting than the binary case since the process trees can be identified with *general trees* with only one type of nodes: an action followed by a set of sub-processes. However, the binary operator used in the present paper is more faithful to the algebraic presentations of process algebras (cf. e.g. [16, 9]). It is well-known that general trees can be encoded in a binary form, hence the change is relatively transparent at the syntax level. The change will be less transparent at the semantic level since we will have to take into account the associative law attached to the parallel operator: $(P_1 \parallel P_2) \parallel P_3 \equiv P_1 \parallel (P_2 \parallel P_3)$. This means, from the semantic view, that process trees that only differ by the left-right succession of parallel nodes must be identified.

2.2 Semantics: computation trees

Process trees are syntax objects that must be interpreted on a *semantic* domain. Our combinatorial model is to interpret process behaviours as *computation trees* [6]. A run or computation is the result of the merging of the branches of a process tree. For example, using our running example from Fig. 1, we note that the run $\langle a, b, c, d, e, f, g\rangle$ is a computation of the process but $\langle a, b, c, d, f, e, g\rangle$ is not because it is not the result of the merging of the process (action f cannot precede action e).

The whole process behaviour is a tree of all possible computations with all common prefixes shared. The right-hand side of Fig. 1 presents the first levels of the computation tree induced by our running example. It is a well-known fact that the computation trees of pure parallel processes are "exponentially" larger than the syntax trees. We can witness this phenomenon of *combinatorial explosion* on our running example. Indeed, despite its small syntactic size (it has 7 counted nodes), its induced computation tree is of size 2360.

The questions that concern us are firstly of a quantitative nature: we would like to give a precise mathematical – in fact combinatorial – meaning for "combinatorial explosion" that is often used in a somewhat gratuitous way. The most significant measure of the process behaviours is undoubtedly their number of runs *on average*. A finer – and technically more involved – measure is required to properly characterize the amount of prefix-sharing in the computation trees. A more qualitative question is then raised: what is the *typical shape* of

[2]The exponential generation function $G(z)$ related to the sequence $(G_n)_n$ satisfies $G(z) = \sum_n G_n \frac{z^n}{n!}$.

the computation trees ? For this we exploit a decomposition of computation trees by levels. A *level* ℓ of a computation tree is the set of nodes that correspond to the ℓ-th occurrence of an action in each of its branches. For example, Fig. 1 depicts the first and second levels of the computation tree of our running example. Ultimately, our theoretical study underlies interesting algorithms for the statistical analysis of computation trees. All these questions shall be now addressed.

3 Typical binary processes

In this section, we are interested in typical measures of computations trees in the context of associative binary parallel processes. We first provide the average asymptotic number of runs of processes of size n, when n tends to infinity. We then refine the quantitative study by considering the total size of the computation trees.

3.1 Typical number of runs

Theorem 3. *The asymptotic of the average number of runs, denoted by $\bar{G}_n$, induced by binary processes of size n, satisfies when n tends to infinity:*

$$\bar{G}_n \sim_{n\to\infty} 3 \cdot \sqrt{\frac{\ln\frac{3}{2}-\frac{1}{3}}{6\sqrt{2}-8}} \cdot \left(\frac{3-2\sqrt{2}}{3\left(\ln\frac{3}{2}-\frac{1}{3}\right)}\right)^n \cdot n!.$$

First remark that $(3-2\sqrt{2})/(3\left(\ln\frac{3}{2}-\frac{1}{3}\right)) \approx 0.79287$ thus the average $\bar{G}_n$ is much smaller than $n!$ that corresponds to the number of runs of the worst processes (where all actions are in parallel).

The proof of this theorem follows the general sketch already followed in [3] for the n-ary process trees. However, our calculations relied, there, extensively on *holonomy theory* and we will see that this approach is not possible in the binary case, we must find other calculation ways. In order to compute the number of runs of a process, we exploit an isomorphism between the runs of a process tree and its *increasing labellings*. Let us recall that usually an increasing labelling of a tree is a labelling of each node with an integer from 1 to the size of the tree such that all successors of a node have a strictly greater label compared to the one of this node. Such so-called *increasing trees* are discussed at length in [8, Chapter 1].

In our model of binary processes, only the nodes labelled by an action must be taken into account. But beyond that the isomorphism still holds.

Lemma 4. *Let P be a binary process. There is an isomorphism between the runs (or computations) of P and the increasing labellings of the nodes of P containing an action.*

The proof for this lemma can be adapted from [3] in a straightforward way. Since parallel nodes are not considered for the increasing labelling, the associativity law of parallel does not play any role here.

On Fig. 2 the two leftmost increasing trees correspond respectively to the runs $\langle a, b, c, d, e, f, g\rangle$ and $\langle a, c, e, f, b, d, g\rangle$ of the running example.

In analytic combinatorics, the *box product* allows to encode increasing constraints on the labels of nodes. Let $\mathcal{A}$ and $\mathcal{B}$ be two combinatorial labelled classes, then the class $\mathcal{A}^{\square} \star \mathcal{B}$ corresponds to labelled objects such that the smallest label belongs to the first component (in $\mathcal{A}$). For further details, see [12, Chapter II].

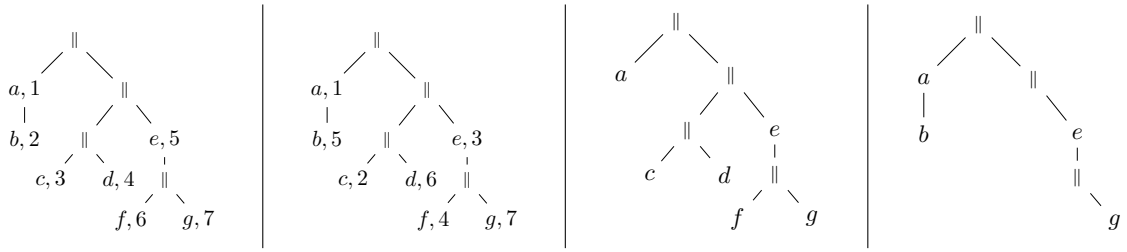

Figure 2: *Two increasingly labelled processes (left) and two admissible cuts (right).*

Proposition 5. *Let $\mathcal{G}$ be the class of increasingly labelled processes, and $G(z)$ its related exponential generating function. We get:*

$$\mathcal{G} = \mathcal{G} \star \mathcal{G} + \mathcal{Z}^{\square} \star (\mathcal{G} + 1), \quad \textit{thus} \quad G(z) = -1 - \frac{3}{2} \cdot \textit{LambertW}\left(-\frac{2}{3}\exp\left(\frac{z-2}{3}\right)\right).$$

The *LambertW*-function satisfies: $\mathrm{LambertW}(z) \cdot \exp(\mathrm{LambertW}(z)) = z$. Many fundamental results about the *LambertW*-function can be found in the paper of Corless *et al* [7]. Especially its generating function is as follows:

$$\mathrm{LambertW}(z) = \sum_{r \geq 1} w_r \; z^r, \qquad \text{where } w_r = \frac{(-r)^{r-1}}{r!}. \tag{1}$$

By taking into account that the series $G(z)$ is exponential, we may easily compute the first numbers of increasingly labelled processes (according to the number of actions): $1, 3, 21, 243, 3933, 81819, \ldots$.

Proof. Due to the translation of the boxed product into the formal power series, we know that the exponential generating function $G(z)$ is the solution, analytic at 0 of $T'(z) - 2T(z)T'(z) - T(z) = 1$ such that $T(0) = 0$ and $T'(0) = 1$. By a partial fraction expansion of this differential equation, we can integrate it and thus we get $\frac{2}{3}(1 + G(z))e^{-\frac{2}{3}(1+G(z))} = e^{\frac{z}{3}-\frac{2}{3}\ln\frac{3}{2}}$ Let us define y and x such that $y = \frac{2}{3}(1 + G(z))$ and $x = e^{\frac{z}{3}-\frac{2}{3}-\ln\frac{3}{2}}$, then the equation turns to $ye^{-y} = x$, thus the link with the *LambertW*-function is exhibited. □ □

As explained previously all the generating functions that we used in [3] where holonomic. Almost all results have been proved by using "Guess and Prove" strategies that rely on holonomicity. As a reminder, a generating function is holonomic if its coefficients satisfy a homogeneous linear (finite) recurrence (called a *P-recurrence*) with polynomials coefficients. From the first coefficients and the P-recurrence generally obtained through guesses followed by cumbersome calculations, one can compute efficiently the next coefficients.

But switching to the binary parallel operator makes the whole *edifice* collapse since the *LambertW*-function is *not* holonomic, as demonstrated in e.g. [13]. From this we can legitimately suppose that the generating function $G(z)$ of Proposition 5 is not holonomic also. However non-holonomy does not always propagate through function composition so that the following is not a trivial result.

Proposition 6. *$G(z)$ is not holonomic.*

This difficulty can be circumvented by following a more direct approach, which is sketched below. We would like to emphasize, however, the important take away of this section: that a binary encoding of an associative operator is not equivalent to its direct interpretation as a general tree. We have holonomy on the one size, and non-holonomy on the other size.

of Theorem 3. The dominant singularity of $G(z)$ is reached when the *LambertW*-function reaches $-e^{-1}$. Hence the dominant singularity of G(z) is $\eta = -1 + 3\ln 3/2$. By basic computations about the *LambertW*-function, we get: $\text{LambertW}(-e^{-1}\cdot(1-h)) =_{h\to 0} -1 + \sqrt{2\,h} + o(\sqrt{h})$. Together with the Taylor development: $-\frac{2}{3}\exp\left(\frac{z-2}{3}\right) = -e^{-1}\cdot\left(1-\frac{\eta}{3}\cdot\left(1-\frac{z}{\eta}\right)\right) + o\left(1-\frac{z}{\eta}\right)$, we get: $\mathit{LambertW}\left(-\frac{2}{3}\exp\left(\frac{z-2}{3}\right)\right) = -1 + \sqrt{\frac{2\eta}{3}\left(1-\frac{z}{\eta}\right)} + o\left(\sqrt{1-\frac{z}{\eta}}\right)$. The classical transfer theorems due to Flajolet and Odlyzko [11], detailed in [12], give:

$$n!\ [z^n]G(z) \sim_{n\to\infty} n!\cdot\frac{3}{2}\cdot\sqrt{\frac{\ln\frac{3}{2}-\frac{1}{3}}{2\pi\ n^3}}\cdot\left(\frac{1}{3\left(\ln\frac{3}{2}-\frac{1}{3}\right)}\right)^n.$$

Finally the stated average value is obtained by normalizing by P_n. □ □

3.2 Typical size of the computation trees

Computation trees share the common computation prefixes, which cannot be witnessed by counting its branches (or leaves) as was done in the previous section. The goal of this subsection is to compute the asymptotic average profile of the computation trees: precisely the average number of nodes of each of their levels.

Theorem 7. *Let $\bar{L}_n$ be the average size of the computations trees induced by binary process trees of size n, and $\bar{L}_n^{n-\ell}$ be the average number of nodes at level $n-\ell-1$ of the computations trees ($\ell \in \{0,\dots n-1\}$). The asymptotic values of these means, when n tends to infinity, satisfies:*

$$\bar{L}_n \sim_{n\to\infty} e\cdot\bar{G}_n, \qquad \textit{and} \qquad \bar{L}_n^{n-\ell} \sim_{n\to\infty} \frac{\bar{G}_n}{\ell!}.$$

Definition 8. *Let P be a process tree. Starting from P, prune iteratively some leaves from the tree structure. If the remaining tree C does not contain leaves labelled by the operator $\|$, then C is called an* admissible cut *of P. The* size *of an admissible cut is the number of actions it contains.*

The complete process tree T is defined as an admissible cut too, but the empty process (after having removed all nodes of T) is not an admissible cut. On Fig. 2, two admissible cuts obtained from our running example (Fig. 1) are depicted.
An increasing labelling of the actions of an admissible cut gives an *increasing admissible cut.*

Lemma 9. *Let P be a process tree. The number of nodes at level $i-1$, for $i \in \{1,\dots,|P|\}$, is equal to the number of increasing admissible cuts of size i of the process P.*

Proposition 10. *The following specification enumerates all increasing admissible cuts induced by process trees of the same size.*

$$\mathcal{C} = \mathcal{C}\times\mathcal{C} + 2\cdot\mathcal{C}\times\mathcal{P} + \mathcal{U}^{\square}\star\mathcal{Z}\times(\mathcal{C}+\mathcal{P}+1),$$

where $\mathcal{Z}$ marks all nodes and $\mathcal{U}$ the nodes of the increasing admissible cuts. Thus,

$$C(z,u) = -(1+P(z)) - \frac{3}{2}\cdot\mathit{LambertW}\left(-\frac{2}{3}\cdot(1+P(z))\cdot\exp\left(\frac{uz}{3}-\frac{2}{3}(1+P(z))\right)\right).$$

An analogous approach as the one presented in the proof of Proposition 5 gives the result.

Proposition 11. *$C(z, u)$ is not holonomic.*

Because of this result, the proof of Theorem 7 is not obvious at all. The result is proved by a detailed analysis of the equation satisfied by $C(z, u)$.

4 Algorithmic applications

The quantitative study described in the previous sections could misleadingly be seen as only of a purely theoretical interest. A better understanding of the average case – or unbiased – situation comes together with a better understanding of the uniform random distribution of the objects under study. This naturally yields interesting algorithmic applications.

In this section we discuss three such applications: (1) the uniform random generation of runs, (2) the computation of the profile of a computation tree, and (3) the covering of computation prefixes at a given process depth. These algorithms take a fixed process tree, say P, as input. In this section we will mostly consider the process of Fig. 3, which is of size 125 and thus with a very large state space (about 10^{145} distinct runs !).

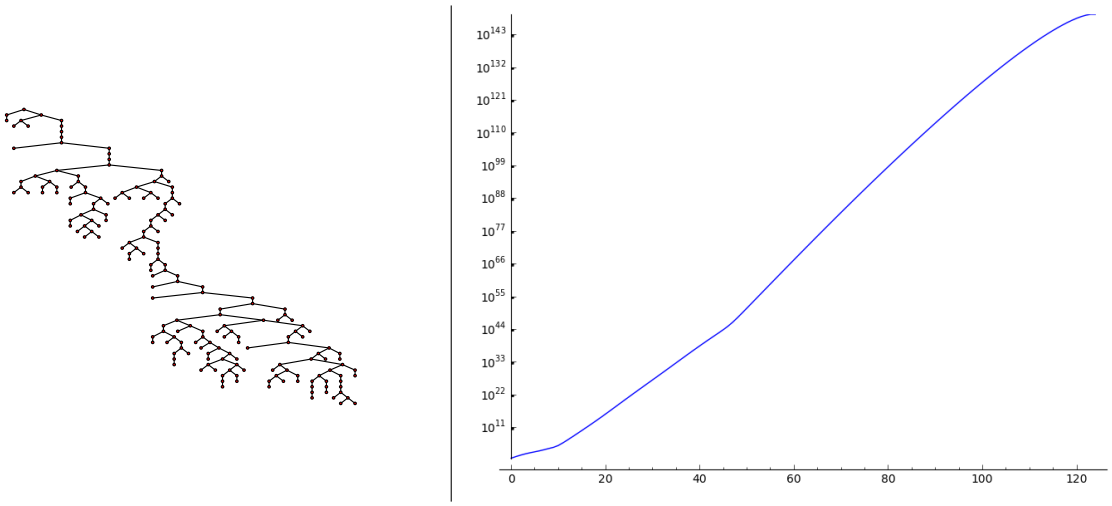

Figure 3: *A process tree of size 125 (left) and the profile of its computation tree (right).*

4.1 Uniform random generation of runs

The uniform random generation of runs is the basic algorithmic building block for the statistical analysis of the process behaviours. In [3] we describe a generation algorithm that works with complexity $O(n \log n)$ with n the size of the initial process tree, described as general trees. Indeed, the random sampler generates branches of the computation trees, distributed uniformly, by only considering the syntax trees, hence avoiding the combinatorial explosion. The algorithm relies on the *hook length formula* [14, P. 67] and the implementation uses a dynamic multiset sampler (the implementation is given as an appendix in [2]).

Luckily enough, the hook length formula can be adapted to the case of the binary parallel processes. For a given process P, let us define a *prefixed subprocess*, P_α as a complete subtree

of P, rooted at an action node α of P (it cannot be rooted in a parallel node). In our running example, P is not a prefixed subprocess of P, because its root is a parallel node, but $P_e := e.(f \parallel g)$ is one.

Proposition 12. *Hook length formula adapatation: Let P be a binary process. The number of computations of P is:*

$$G_P = \frac{|P|!}{\prod_{P_\alpha \text{ prefixed subprocess}} |P_\alpha|}.$$

This proposition is direct once we remark that partially increasing trees of P (only actions are increasingly labelled) are in one-to-one correspondence with runs of P.

Corollary 13. *Let P be a process of size n. Using the hook length formula adaptation, we derive an uniform random generator to sample runs in P. This time complexity is $O(n \log n)$.*

The algorithm is an adaptation of the sampler obtained in [3] based on the multiset sampler. The time complexity is not impacted by the parallel nodes since their number is bounded by n.

4.2 Profile of the computation tree

Using the hook length formula adaptation, we get an efficient way to compute the number of nodes of the last level of the computation tree, hence its number of runs. The time complexity of the algorithm is linear in the size of the process tree. However we are not just interested by the number of leaves of the tree, but by its whole *profile* i.e. the number of nodes at each of its levels. This is the qualitative facet of our level decomposition of computation trees, introduced in Section 3. In [5] we develop a naive algorithm to compute the profile of a computation tree. This algorithm has exponential time complexity and can thus only be applied on small process trees. In an experiment we were able to compute the profile of a process tree of size 40 in "a couple of days using a fast parallel computer". We now introduce a much more efficient algorithm (its complexity is linear!). Indeed, the profile of the computation tree corresponding to the process tree of size 125 of Fig. 3 is constructed in less than a second using Sage/Python on a basic PC. Recall that the total size of the computation tree is about 10^{145}. The profile is presented on Fig. 3 with a semilogarithm scale for the number of nodes by level.

In practice, the profile algorithm can be seen as a by-product of a more general algorithm that allows us to sample prefixes of computations uniformly among all the prefixes of the same sizes. It is based on the *recursive method* developed by Nijenhuis and Wilf [18]. It is interesting to note that the algorithm we implemented is very close to (indeed inspired by) the one we develop in [4] for a very different purpose: the uniform random generation of computations for non-deterministic processes. The fundamental idea of the algorithm is to see a process tree as the combinatorial class of all its admissible cuts. To define such a class, we need to introduce two concepts in the process specification. First, the empty subprocess ϵ and then an operator, denoted by $+$, that allows to choose between a prefixed subprocess or this empty subprocess. Thus the subprocess $P_\alpha + \epsilon$ consists of running the subprocess P_α or of stopping the process. Obviously, this $+$ operator has no relation to the nondeterministic choice of [4], although, from an analytic combinatorics view both play the same rôle. In the context of [4], the algorithm allows to generate runs with nondeterministic choice, but it has no relation with the profile problem we address here. Thus, both algorithms are based on

some similar polynomial, but their interpretation is completely different. The algorithm's sketch is as follows: For a given process P:

- Define the specification of the combinatorial class of all its admissible cuts: i.e. replace each prefixed subprocess P_α by $(P_\alpha + \epsilon)$.
- Compute the polynomial representation of this combinatorial class: it corresponds to the exponential generating function of the admissible cuts class.
- Using the polynomial representation, one can easily:
  - Compute the computation tree profile.
  - Sample computations prefixes or complete computation according to this distribution by using the recursive method.

By using a directed acyclic graph for the representation of the polynomial, the profile computation is obtained in $O(n)$ time complexity; and the random sampling in $O(n \log n)$. Both complexity results are derived from the proofs of [4].

4.3 Prefix covering

The previous algorithm allows us to construct the uniform probability distribution on run prefixes of a given length. This opens up an interesting statistical analysis approach that is complementary to the generation of complete runs. It is indeed difficult to talk about *covering* when generating complete runs. In all but the most trivial situations there are simply too many possible runs to even think about covering the full process behaviour. The only guarantee provided is that the sampled computations are truly random and not biased in one way or another. An interesting covering criterion is that of the *process depth*. We now discuss such a covering algorithm that can generate computation *prefixes* of a given length – corresponding to the expected process depth – and uniformly at random. The algorithm finds its justification in the "coupons collector" principle (cf. [10]). When sampling objects whose probability distribution is uniform, then the expected number of samples necessary to collect all the object is equivalent to $n \log n$. This is the smallest possible expectation since the distribution is uniform.

Another covering method would consist in following a *local uniform probability distribution* instead of the global – in fact the *real* – one. When sampling locally uniformly, we only treat each branch of a parallel construct with equal probability, without taking into account the past and the future of the behaviour. But even compared with this naive and deficient approach, it is largely outperformed by our covering algorithm for the global distribution.

For the process of Fig. 3, we conducted some experiments:

Prefix length	1	2	3	4	5	6	7	8
Nb of prefixes	4	14	43	115	265	564	1201	2877
Expected time for covering								
Local uniformity	8	47	223	972	4343	24087	137174	914313
Global uniformity	8	45	187	612	1646	3891	8993	21719
Gain	0%	4.5%	19%	59%	164%	519%	1425%	4110%

The prefixes covering is always much more affordable using our global technique. The larger the prefix length is, the more our approach is unavoidable.

References

[1] F. Bassino, J. David, and C. Nicaud. Average case analysis of Moore's state minimization algorithm. *Algorithmica*, 63(1-2):509–531, 2012.

[2] F. Bergeron, P. Flajolet, and B. Salvy. Varieties of increasing trees. In *CAAP*, pages 24–48, 1992.

[3] O. Bodini, A. Genitrini, and F. Peschanski. Enumeration and random generation of concurrent computations. In *DMTCS AofA'12 proceedings*, pages 83–96, 2012.

[4] O. Bodini, A. Genitrini, and F. Peschanski. The Combinatorics of Non-determinism. In *proc. FSTTCS 2013*, volume 24 of *(LIPIcs)*, pages 425–436, 2013.

[5] O. Bodini, A. Genitrini, and F. Peschanski. Enumeration and random generation of concurrent computations. In *arXiv/1407.1873*, page Under Submission, 2014.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[7] R. M. Corless, G. H. Gonnet, D. E. G. Hare, D. J. Jeffrey, and D. E. Knuth. On the Lambert W Function. In *Advances in Computational Mathematics*, volume 5, pages 329–359, 1996.

[8] M. Drmota. *Random trees.* Springer, Vienna-New York, 2009.

[9] G. Duchamp, F. Hivert, J.-C. Novelli, and J.-Y. Thibon. Noncommutative Symmetric Functions VII: Free Quasi-Symmetric Functions Revisited. *Ann. Comb.*, 15:655–673, 2011.

[10] P. Flajolet, D. Gardy, and L. Thimonier. Birthday Paradox, Coupon Collectors, Caching Algorithms and Self-Organizing Search. *D. A. Math.*, 39(3):207–229, 1992.

[11] P. Flajolet and A. M. Odlyzko. Singularity analysis of generating functions. *In SIAM J. Discrete Math.*, 3:216–240, 1990.

[12] P. Flajolet and R. Sedgewick. *Analytic Combinatorics.* Cambridge UP., 2009.

[13] S. Gerhold. On Some Non-Holonomic Sequences. *Elec. J. Comb.*, 11(1):1–8, 2004.

[14] D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching.* Addison Wesley Longman Publishing Co., Inc., 1998.

[15] D. Krob, J. Mairesse, and I. Michos. Computing the average parallelism in trace monoids. *Discrete Mathematics*, 273(1-3):131–162, 2003.

[16] R. Milner. *A Calculus of Communicating Systems.* Springer Verlag, 1980.

[17] M. Mishna and M. Zabrocki. Analytic aspects of the shuffle product. *CoRR*, 2008.

[18] H. S. Wilf and A. Nijenhuis. *Combinatorial algorithms : An update.* 1989.