Statistical Analysis of Non-Deterministic Fork-Join Processes

Martin Pépin, Antoine Genitrini, Frédéric Peschanski

Statistical Analysis of Non-Deterministic Fork-Join Processes

Martin Pépin, Antoine Genitrini, and Frédéric Peschanski

Sorbonne Université, LIP6, CNRS UMR7606
{martin.pepin,antoine.genitrini,frederic.peschanski}@lip6.fr

Abstract. We study the combinatorial structure of the state-space of non-deterministic fork-join processes. As a first step we establish a link between concurrent programs and a class of combinatorial structures based on the notion of increasing labelling. Beyond the theory, our goal is to develop algorithms and tools for the statistical analysis of the process behaviours. In the second part we develop and experiment two efficient random sampling algorithms serving as basic building blocks for state-space exploration. One is a uniform random sampler of bounded executions, providing a good default exploration strategy, and the other is a uniform sampler of execution prefixes allowing to bias the exploration in a controlled manner. The fundamental characteristic of these algorithms is that they work on the control graph of the programs and do not require the construction of its state-space, thus providing a way to tackle the infamous state explosion problem.

1 Introduction

Analyzing the state-space of concurrent programs is a notoriously difficult task, if only because of the infamous *state explosion* problem. Several techniques have been developed to "fight" this explosion: symbolic encoding of the state-space, partial order reductions, exploiting symmetries, etc. An alternative approach is to adopt a probabilistic point of view, for example by developing *statistical* analysis techniques such as [14]. The basic idea is to generate random executions from program descriptions, sacrificing exhaustiveness for the sake of tractability. However, there is an important difference between generating an *arbitrary* execution and generating a random execution in a *controlled* manner since the later gives a better understanding and a better estimation of the coverage of the state-space, for which the uniform distribution plays an important role.

As a preliminary, we have to find suitable combinatorial interpretations for the fundamental constructions of concurrent programs. In this paper, we study a class of programs that uses a *fork-join* model of synchronization, together with loops and a choice construct for *non-determinism*. This is a simple formalism, but it is non-trivial in terms of the concurrency features it provides. Most importantly, the underlying combinatorial interpretation is already quite involved. The work presented in this paper builds on, and synthesizes, previous studies of "pure" parallelism [8], non-determinism [7] and synchronization [9]. Of course,

the separate analysis of the parts does not tell everything about the *whole*, and the integration of a loop construction in a new contribution. If compared to [7], in particular, we use a more direct and simple encoding of non-deterministic choices, which significantly improves both the theoretical and practical developments. The algorithms presented in the paper are in consequence much more efficient in practice, while covering a more expressive language.

The first basic problem we study is that of *counting* the number of executions of bounded length of the programs. This measure allows to quantify the state-space explosion. Unfortunately, counting executions of even simple programs is in fact hard. For instance, we show in [9] that programs that can only perform atomic actions and *barrier synchronizations* are expressive enough so that counting executions is a $\sharp P$-complete problem (see details in [2]). Fork-join parallelism enables a good balance between tractability and expressivity by enforcing some structure in the state-space. A second problem is caused by non-determinism, because for each (local) choice we have to select a unique branch of execution, and choices may be nested so that the number of possibilities can grow exponentially. Relying on an efficient encoding of the state-space as generating functions we manage to count executions without expanding the choices.

Of course counting executions has no direct practical application, but it is an essential requirement for two complementary and more interesting analysis techniques we discuss in the paper. We build on top of well-known algorithms from combinatorics [13,20] to develop first, a uniform random sampler of executions of given length and second, a uniform random sampler of execution *prefixes*. Both algorithms offer a *controlled* way to explore the state-space where "controlled" means that we know the distribution of the sampled objects. We see the first algorithm as a "default" exploration strategy since, without prior knowledge of the state-space, it gives the best coverage, with the best diversity of outputs. Since "uniform" does not make sense in presence of an infinite state-space, sorting the executions by length is a way to still consider finite sets and to offer uniformity inside these sets. Thus our uniform-by-size sampler can be used in conjunction with a strategy for picking the size, for instance to generate uniform executions of bounded length. The second algorithm generates random prefixes, which allows the user to introduce *bias* in the statistical exploration strategy, e.g. towards regions of interest of the state-space, while still giving a good coverage and most importantly still giving control over the distribution. A fundamental characteristics of these algorithms is that they work on the syntactic representation of the program and do not require the explicit construction of the state-space, hence enabling the analysis of systems of a rather large size.

The outline of the paper is as follows. In Section 2 we present the program class of non-deterministic fork-join programs, as well as its combinatorial interpretation. In Section 3 we describe an efficient algorithm for the uniform random generation of executions and execution prefixes of given length. Finally, Section 4 provides a preliminary experimental study of the algorithms[1].

[1] An implementation of our algorithms and all the scripts used for the experiments can be found on the companion repository at https://gitlab.com/ParComb/libnfj.

Related work Our study combines viewpoints and techniques from concurrency theory and combinatorics. A similar line of work exists for the so-called "true concurrency" model (by opposition to our interleaving semantics) based on the trace monoid using *heaps combinatorics* (see [18,1]). In [3], the authors cover the problem of the uniform random generation of words in a class of synchronised automata called "synchronised automata". This approach is able to cover a slightly more expressive set of programs but this comes at the cost of the construction of an automaton of exponential size. Another approach, investigated in the context of Monte-Carlo model-checking, is based on the combinatorics of *lassos*, which relates to the verification of some temporal-logic properties over potentially *infinite* executions In [15], the authors of this method highlight the importance of uniformity. Later [21] gives a uniform random sampler of *lassos*, however relying on the costly explicit construction of the whole state-space, hence unpractical for even small processes. Finally [10] studies the random generation of executions in a model similar the one we cover by extending the framework of Boltzmann sampling. Although Boltzmann samplers are usually very fast, they turn out to be slow in this context because of the heavy symbolic computations imposed by the interplay between parallelism and synchronisation.

2 Non-deterministic fork-join programs with loops

Here we introduce a simple class of concurrent programs featuring a Unix-like fork-join programming style with non-deterministic choices and loops. The interest is twofold. First it showcases non-determinism in interaction with a non-trivial programming model which gives insights about its quantitative and algorithmic aspects. Second it has a loop construction for which we give a combinatorial interpretation, which is an important step forward in terms of expressivity compared to our previous work (see [8,7,5] for instance). Throughout the paper we will refer to this class as the class of *non-deterministic fork-join programs*.

2.1 Syntax

Definition 1 (Non-deterministic fork-join programs). *Given a set of symbols $\mathcal{A}$ representing the "atomic actions" of the language, the class of non-deterministic fork-join programs (over this set $\mathcal{A}$), denoted NFJ, is defined as follows:*

$$
\begin{array}{rll}
P, Q & ::= P \parallel Q & \textit{(parallel composition (or fork))} \\
& \mid P; Q & \textit{(sequential composition (or join))} \\
& \mid P + Q & \textit{(non-deterministic choice)} \\
& \mid P^{\star} & \textit{(loop)} \\
& \mid a \in \mathcal{A} & \textit{(atomic action)} \\
& \mid 0 & \textit{(empty program, noop).}
\end{array}
$$

Informally, the first two constructions form the fork-join "core" of the language: $P \parallel Q$ expresses the fact that P is run in parallel with Q and $P; Q$ means

that P must terminate before Q starts. In $a; (b \parallel (c; d)); e$, the program starts by firing a, then it *forks* two processes b and $c; d$ which run in parallel and when they terminate e is run, which is called a "join". The third construction $P + Q$ expresses a *choice*: either P or Q is executed but not both, this is typically an `if` in programming languages. This can model an "internal" choice of the system such as a random event, a system failure etc, or an "external" choice, that is a choice depending on a user input. Finally, the construction $P^\star$ expresses loops that can have any (finite) number of iterations. For instance $a; (b \parallel c)^\star$ can be expanded to 0 (zero iteration), $a; (b \parallel c)$ (one iterations), $a; (b \parallel c); a; (b \parallel c)$ (two iterations), etc.

It is important to mention now that the nature of the atomic actions will remain abstract in the present work, we treat them as black boxes and will consider that the different occurrences of an action across a term are distinct. They are sometimes referred to as *events* in the literature. Our focus is set on the order in which these actions can be fired and scheduled by the different operators of the language. In all our examples we use a different lowercase roman letters as a unique identifiers to help distinguishing between each action.

This simple model is expressive enough to write simple programs in the fork-join style (like the way Unix processes work). Moreover, the four combinators present in the grammar above can be modelled and well-understood using the tools from analytic combinatorics, which is at the core of our random sampling procedure in Section 3.

2.2 Semantics

We give NFJ an operational semantics in the style of [17]. The "step" relation $P \xrightarrow{a} P'$ between two programs and an atomic action describes one atomic computation step. It reads "program P reduces to P' by firing action a". The idea behind the rules is explain just below.

$$\frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q}\ (Lpar) \qquad \frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'}\ (Rpar) \qquad \frac{P \xrightarrow{a} P'}{P; Q \xrightarrow{a} P'; Q}\ (Lseq)$$

$$\frac{\text{nullable}(P) \quad Q \xrightarrow{a} Q'}{P; Q \xrightarrow{a} Q'}\ (Rseq) \qquad \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}\ (Lchoice) \qquad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}\ (Rchoice)$$

$$\frac{}{a \xrightarrow{a} 0}\ (act) \qquad \frac{P \xrightarrow{a} P'}{P^\star \xrightarrow{a} P'; P^\star}\ (loop)$$

Where the nullable predicate, defined just bellow, tells whether a program can terminate without firing any action.

$$\begin{aligned}
&\text{nullable}(P \parallel Q) = \text{nullable}(P) \wedge \text{nullable}(Q) && \text{nullable}(0) = \top \\
&\text{nullable}(P; Q) = \text{nullable}(P) \wedge \text{nullable}(Q) && \text{nullable}(a) = \bot \\
&\text{nullable}(P + Q) = \text{nullable}(P) \vee \text{nullable}(Q) && \text{nullable}(P^\star) = \top
\end{aligned}$$

The rules for the parallel composition (*Lpar* and *Rpar*) express the interleaving semantic of the language: if an action can be fired in any of P or Q, then it can be fired in $P \parallel Q$ and the term is rewritten. By iterating these two rules, we can obtain any interleaving of an execution of P and an execution of Q. Sequential composition is more asymmetric. The *Lseq* rule is similar to *Lpar* but *Rseq* captures the synchonisation: an execution can be fired on the right-hand-side only if the left-hand-side is ready to terminate (expressed by nullable(P)), in which case it is erased. The choice rules *Lchoice* and *Rchoice* allow actions to be fired from both sides but once we have made the choice of the branch, it is made definitive by erasing the other branch. Finally the loop $P^\star$ can either expand to 0, which cannot fire any action or expand to $(P; P^\star)$. The rule *loop* corresponds to applying *Lseq* to $(P; P^\star)$. The executions of the language are defined as a sequences of steps leading to a nullable term.

Definition 2 (Execution). *An execution of an NFJ program P is a sequence of reduction steps of the form $P \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 \dots \xrightarrow{a_n} P_n$ such that* nullable(P_n) *holds.*

Remark 1 (on equality). Two reductions obtained via different proof-trees are different. For instance there are two distinct executions depicted by $a^{\star\star} \xrightarrow{a} (0; a^\star; a^{\star\star}) \xrightarrow{a} (0; a^\star; a^{\star\star})$. One corresponds to the case where the outer loop is only expanded once but the inner loop twice. The other corresponds to the opposite.

We will take the following program as a running example for the rest of the paper: $P_0 = ((a + (b \parallel c))^\star \parallel (d + 0))^\star; (e + (f \parallel g))$. For instance, the shortest execution of this program is $P \xrightarrow{e} 0$, obtained by applying the *(Rseq)* and *(Lchoice)* rules, and its four length-2 executions are:

- $P_0 \xrightarrow{f} (0 \parallel g) \xrightarrow{g} 0 \parallel 0$
- $P_0 \xrightarrow{g} (f \parallel 0) \xrightarrow{f} 0 \parallel 0$
- $P_0 \xrightarrow{a} ((0; (a + (b \parallel c))^\star) \parallel (d + 0)); ((a + (b \parallel c))^\star \parallel (d + 0))^\star; (e + (f \parallel g)) \xrightarrow{e} 0$
- $P_0 \xrightarrow{d} ((a + (b \parallel c))^\star \parallel 0); ((a + (b \parallel c))^\star \parallel (d + 0))^\star; (e + (f \parallel g)) \xrightarrow{e} 0.$

2.3 Combinatorial interpretation

We give here an interpretation of the executions of an NFJ program as combinatorial objects, which will open us the toolbox of analytic combinatorics for the rest of the paper. We model the set of the executions of a program as a *combinatorial class* using the formalism of *labelled classes* from [12], which we recall here. A combinatorial class is a potentially infinite set of objects where each object has been given a (finite) size and in which there is only a finite number of objects of each size. In our case, the combinatorial class of interest is the set of executions of a given program and the size of an execution is its length i.e. its number of reduction steps.

A *labelled* class is a combinatorial class where objects are given with a labelling i.e. a permutation of the same size as the object. Often this permutation

is seen as a labelling of some component of the object. In our case, it will encode the *scheduling* of the actions of the program. A more in-depth introduction to the topic of combinatorics of labelled and unlabelled classes can be found in the two first chapters of the book [12].

The combinatorial class $S(P)$ modelling the executions of P is inductively defined in Table 1. The explanations for the combinatorial constructions are given bellow.

Table 1: Recursive rules for the computation of the generating function of executions of an NFJ program.

Construction	Specification
P	$S(P)$
0	$\mathcal{E}$
a	$\mathcal{Z}$
$P \parallel Q$	$S(P) \star S(Q)$
$P; Q$	$S(P) \boxtimes S(Q)$
$P + Q$ when nullable$(P) \wedge$ nullable(Q)	$S(P) + (S(Q) \setminus \mathcal{E})$
$P + Q$ otherwise	$S(P) + S(Q)$
$P^\star$ when nullable(P)	$\textsc{Set}^{\boxtimes}(S(P) \setminus \mathcal{E})$
$P^\star$ otherwise	$\textsc{Set}^{\boxtimes}(S(P))$

The empty program 0 and the atomic action a have only one execution, of length 0 and 1 respectively. This is modelled combinatorially by the neutral class $\mathcal{E}$: the class containing only one element of size 0, and the atom class $\mathcal{Z}$: the class with only one element of size 1.

The first interesting case is the parallel composition: the executions of $P \parallel Q$ are made of any interleaving of one execution of P and one execution of Q. The *labelled product* of combinatorics expresses exactly this and is denoted using the $\star$ symbol. Said differently, *scheduling* two parallel executions maps combinatorially to *shuffling* two labellings. An execution of $P; Q$ is given by an execution of P followed by an execution of Q. To model this, we use the *ordered product* of combinatorics, denoted $\boxtimes$ which is an analog of the labelled product without the interleaving of the labellings (see [6]). The set of executions of $P + Q$ is the union of the executions of P and Q. Moreover this union is "almost" disjoint in the sense that the only execution that these programs may have in common is the empty execution, hence the two case in the definition. Combinatorially, the fact that nullable(P) holds corresponds to the fact that the class of its execution contains one object of size 0: the empty execution. It is in fact important that we can express this in terms of *disjoint* unions because they fit in the framework of analytic combinatorics whereas arbitrary unions are more difficult to handle[2].

Finally, the executions of $P^\star$ are sequences of executions of P or, equivalently, sequences of *non-empty* executions of P. This second formulation leads to a non-ambiguous specification as the unique class $\mathcal{P}'$ satisfying $\mathcal{P}' = \mathcal{E} + \mathcal{P}_+ \boxtimes \mathcal{P}'$, where $\mathcal{P}_+$ denotes the non-empty executions of P. This implicitly defined class $\mathcal{P}'$

[2] Grammar descriptions involving non-disjoint unions are referred to as "ambiguous" and lack most of the benefits, if not all, of the symbolic method, essentially because some objects may be counted multiple times when applying the method.

is denoted $\textsc{Set}^{\boxed{\star}}(\mathcal{P}_+)$ and is called the *ordered set* of $\mathcal{P}_+$ (see [6]). Once again we must distinguish whether nullable(P) or not in the definition of $\mathcal{P}_+$ to avoid ambiguity and thus double-counting.

The S function described above maps each program to a combinatorial specification of its executions. As an example, for our example program we have $S(P_0) = \textsc{Set}^{\boxed{\star}}(\textsc{Set}^{\boxed{\star}}(\mathcal{Z} + (\mathcal{Z} \star \mathcal{Z})) \star (\mathcal{Z} + \mathcal{E}) \setminus \mathcal{E}) \boxed{\star} (\mathcal{Z} + (\mathcal{Z} \star \mathcal{Z}))$. Such a specification is often the starting point of the study of a problem in analytic combinatorics, because it has many outcomes, and the most important one of them is that it gives a systematic way to compute the generating function of the combinatorial class. We recall that the generation function of a class $\mathcal{C}$ is the formal power series given by $C(z) = \sum_{n \geqslant 0} c_n z^n$ where c_n is the number of elements of size n in $\mathcal{C}$.

The generating function of the executions of a program, i.e. of the class $S(P)$, constitutes a summary of the counting information of the state space. Moreover, this encoding as a power series gives a convenient formalism to compute the sequence from a combinatorial specification. The *symbolic method* from [12] gives an automatic translation from the specification to generating function, which we recall in Table 2.

Specification $\mathcal{A}$	Gen. Function $A(z)$
$\mathcal{E}$	1
$\mathcal{Z}$	z
$\mathcal{A} \setminus \mathcal{B}$	$A(z) - B(z)$
$\mathcal{A} + \mathcal{B}$	$A(z) + B(z)$
$\mathcal{A} \star \mathcal{B}$	$A(z) \circledcirc B(z)$
$\mathcal{A} \boxed{\star} \mathcal{B}$	$A(z) \cdot B(z)$
$\textsc{Set}^{\boxed{\star}}(\mathcal{A})$	$(1 - A(z))^{-1}$

Table 2: The rules of the symbolic method for computing a generating function from a combinatorial specification. In the case of the labelled product $\mathcal{A} \star \mathcal{B}$ the corresponds to the *coloured product* $\circledcirc$, defined in [6] by $A(z) \circledcirc B(z) = \sum_{n \geqslant 0} \sum_{k=0}^{n} \binom{n}{k} a_k b_{n-k} z^n$. In the case of the set difference, $\mathcal{B}$ must be a subset of $\mathcal{A}$.

Remark 2. the reader familiar with combinatorics might find it odd that we use *ordinary* generating functions (OGF) rather than exponential ones (EGF) which are generally more suitable for labelled setups. In fact we are dealing with a mix of both worlds, thus a choice is needed. We opt for the OGF setup for efficiency reasons in what follows.

The aim of this last paragraph is to demonstrate the power of these tools by showing how a few manipulations on polynomials can yield to interesting algorithmic applications and precise quantitative results. Further resource on this topic can be found in the book [12]. We study the generating function f of the example program P_0 given above. Let $f(z) = \sum_{n \geqslant 0} p_n z^n$ denote its expansion in power series and recall that the n-th coefficient p_n is the number of executions of P_0 of length n. By applying the rules from Table 2 to $S(P_0)$ we obtain that $f(z) = \left(2 - (1 - z - 2z^2)^{-1} \circledcirc (z + 1)\right)^{-1} \cdot (z + 2z^2)$. Using the

rule[3] $z \circledcirc A(z) = z\frac{d(zA(z))}{dz}$ leads to $\frac{(2z+1)(2z-1)^2(z+1)^2 z}{1-4z-4z^2+6z^3+8z^4}$. From this formula we derive two applications. First, from this rational expression we deduce that for all $n > 6$ we have $p_n - 4p_{n-1} - 4p_{n-2} + 6p_{n-3} + 8p_{n-4} = 0$. The obtained recurrence formula can be used to compute the number of executions of length n in linear time. On the analytic side, f being a rational function, we can do a *partial fraction decomposition* to obtain f as a sum of four terms of the form $C_i(1-z\rho_i^{-1})$ (plus a polynomial). Each of these terms expands as $\sum_{n\geqslant 0} C_i \rho_i^{-n} z^n$, hence the number of executions of P_0 of length n satisfies $p_n = C \cdot \rho^{-n} \cdot (1+o(1))$ for some constants C and ρ. In this case $\rho \approx 0.221987$, $C \approx 0.146871$ and the error term hidden in the $o(1)$ is of the order of 0.327950^n.

3 Statistical analysis algorithms

In this section, we consider given an NFJ program and we study the problem of exploring its state-space through random generation. To this end, we will describe two algorithms, a uniform random sampler of executions of given length and a uniform random sampler of execution prefixes. Our approach relies on the counting information contained in the generating functions.

3.1 Preprocessing: the generating function of executions

As explained in the previous section, the symbolic method gives a systematic way of computing the generating functions from the specification $S(P)$ of the executions of the program using the rules from Table 2. A straightforward application of this method leads to Algorithm 1 for computing the first terms of the series.

Algorithm 1 Computation of the generating function of the executions of an NFJ program up to degree n

Input: An NFJ program P and a positive integer n.
Output: The first $n+1$ terms of the generating function of P

```
function GFUN(P, n)
    if P = 0 then return 1
    else if P = a then return z
    else if P = Q ∥ R then return GFUN (Q, n) ⊛ GFUN (R, n) mod z^{n+1}
    else if P = Q; R then return GFUN (Q, n) · GFUN (R, n) mod z^{n+1}
    else if P = Q + R then
        q ← GFUN (Q, n),    r ← GFUN (R, n)
        if q(0) = r(0) = 1 then return q + r − 1 else return q + r
    else if P = Q* then
        q ← GFUN (Q, n)
        return (1 − (q − q(0)))^{−1} mod z^{n+1}
```

[3] this is the only "non-standard" computation rule we use in this example. All the rest is usual polynomial manipulations. General rules for computing $A(z) \circledcirc B(z)$ are beyond the scope of this article.

The coloured product $\circledcirc$ used in the parallel composition case can be implemented using the formula $A(z) \circledcirc B(z) = \mathrm{Lap}(\mathrm{Bor}(A) \cdot \mathrm{Bor}(B))$ where Bor and Lap are respectively the combinatorial Borel and Laplace transforms (see [6]). This approach has the advantage of benefiting from the efficient polynomial multiplication algorithms from the literature at the cost of three linear transformations. To be implemented efficiently, the coefficients of the result of the Borel transform should share $n!$ as a common denominator so that it is only stored once and we keep working integer coefficients. The computation of $(1 - A(z))^{-1}$ can be carried efficiently using Newton iteration (see [22] for instance). The idea is to iterate the formula $S_{i+1}(z) \leftarrow S_i(z) + S_i(z) \cdot (A(s) \cdot S_i(z) - (S_i(z) - 1))$, starting from $S_0(z) = 1$. It has been shown that only $\lceil \log_2(n+1) \rceil$ iterations are necessary for the coefficients of $S_i(z)$ to be equal to those of $(1 - A(z))^{-1}$ up to degree n. Moreover the total cost of this procedure in terms of integer multiplication is of the same order of magnitude as that of the multiplication of two polynomials of degree n.

Theorem 1. *Let P an NFJ program and let $|P|$ denote its number of constructors. Algorithm 1 can be implemented to compute the first n coefficients of the generating function of the executions of P in $O(|P|M(n))$ operations on big integers where $M(n)$ is the complexity of the multiplication of two polynomials. These coefficients are bounded by $n!$ and hence have at most $n \ln(n)$ bits.*

Proof. The proof of this theorem follows from the above discussion: each constructor incurs one polynomial operation among addition, multiplication, coloured product and inversion and all of them can be carried in $O(M(n))$.

3.2 Random sampling of executions

Another consequence of having a combinatorial specification of the state-space is that we can apply well-known random sampling methods from the combinatorics toolbox. Our random sampling procedure for program executions is based on the so-called "recursive-method" from [13]. It operates in a similar fashion to the symbolic method, that is by induction on the specification by combining the random samplers of the sub-structures with simple rules depending on the grammar construction. For the sake of clarity we represent executions as sequences of atomic actions. This encoding does not contain all the information that defines an execution, typically it does not reflects in which iteration of a loop an atomic action is fired for instance. However it makes the presentation clearer. The algorithm can be easily adapted to another more faithful encoding. Our uniform random sampler of executions is described in Algorithm 2 and the detailed explanations about the different construction are given below.

Choice The simplest rule of the recursive method is that of the disjoint union used at line 4 of Algorithm 2. If q_n and r_n denote the number of length-n executions of Q and R, then a uniform random length-n execution of $P = Q + R$ is a uniform length-n execution of Q with probability $q_n/(q_n + r_n)$ and a uniform

Algorithm 2 Uniform random sampler of executions of given length

Input: A program P and an integer n such that P has length n executions.
Output: A list of atomic actions representing an execution

```
1: function UnifExec(P, n)
2:     if n = 0 then return the empty execution
3:     else if P = a then return a
4:     else if P = Q + R then
5:         if Bernoulli (q_n / (q_n + r_n)) then return UnifExec (Q, n)
6:         else return UnifExec (R, n)
7:     else if P = Q ∥ R then
8:         draw k ∈ ⟦0; n⟧ with probability (n choose k) q_k r_{n-k} / p_n
9:         return shuffle (UnifExec (Q, k), UnifExec (R, n − k))
10:    else if P = Q; R then
11:        draw k ∈ ⟦0; n⟧ with probability q_k r_{n-k} / p_n
12:        return concat (UnifExec (Q, k), UnifExec (R, n − k))
13:    else if P = Q* then
14:        draw k ∈ ⟦1; n⟧ with probability q_k p_{n-k} / p_n
15:        return concat (UnifExec (Q, k), UnifExec (P, n − k))
```

The lower case letters p_n, q_k, r_{n-k} etc. indicate the number of executions of length n, k, etc. of programs P, Q and R.

length-n execution of R otherwise. One way to draw the Bernoulli variable is to draw a uniform random big integer x in $[\![0; q_n + r_n[\![$ and to return `true` if and only if $x < q_n$. As an example, consider the programs $Q = (a + (b \parallel c))$ and $R = d^\star$. We count that Q has two executions of length two: bc and cb and R has only one: dd. Hence, to sample a length-2 execution in $(Q + R)$, one must perform a recursive call on Q with probability $2/3$ and on R with probability $1/3$.

Parallel composition The other rules build on top of the disjoint union case. For instance, the set of length-n executions of $P = Q \parallel R$ can be seen as $\mathcal{Q}_0 \star \mathcal{R}_n + \mathcal{Q}_1 \star \mathcal{R}_{n-1} + \cdots + \mathcal{Q}_n \star \mathcal{R}_0$ where $\mathcal{Q}_k$ (resp. $\mathcal{R}_k$) denotes the set of length-k executions of Q (resp. R). By generalising the previous rule to disjoint unions of $(n + 1)$ terms, and using the fact that the number of elements of $\mathcal{Q}_k \star \mathcal{R}_{n-k}$ is $q_k r_{n-k} \binom{n}{k}$, one can select in which one of these terms to sample by drawing a random variable which is k with probability $q_k r_{n-k} \binom{n}{k} / p_n$. Then it remains to sample a uniform element of $\mathcal{Q}_k$, a uniform element of $\mathcal{R}_{n-k}$ and a uniform shuffling of their labellings among the $\binom{n}{k}$ possibilities. This is described at line 7 of Algorithm 2. We do not detail the implementation of the shuffling function here, an optimal algorithm in terms of random bits consumption, can be found in [5]. As an example, consider the same programs as above: $Q = (a + (b \parallel c))$ and $R = d^\star$. The number of length-3 executions of $(Q \parallel R)$ is $1 \cdot 1 \cdot \binom{3}{1} + 2 \cdot 1 \cdot \binom{3}{1} = 9$ using the decomposition $Q_1 \star R_2 + Q_2 \star R_1$. Say $k = 1$ is selected (with probability $1/3$), then the recursive calls to $(Q, 1)$ and $(R, 2)$ necessarily return a and dd and the SHUFFLE procedure must choose uniformly between add, dad and dda.

Sequential composition The case of the sequential composition is similar (see line 10 of Algorithm 2). We use the same kind of decomposition, using the ordered product $\boxtimes$ in place of the labelled product $\star$. This has the consequence of removing the binomial coefficient in the formula for the generation of the k

random variable. Once k is selected, we generate an execution of Q_k, an execution of R_{n-k} and we concatenate the two.

Loop Finally, the case of the loop is a slight adaptation of the case of the sequential composition using the fact that the executions of $Q^\star$ are the executions of $(0 + Q; Q^\star)$. However, care must be taken to avoid issues related to double-counting. More specifically, when sampling an execution of $(Q; Q^\star)$ we must not choose an execution of length 0 for the left-hand-side Q. This is related to the same reason we had to specify the executions of $Q^\star$ as all the sequences of *non-empty* executions of Q. This is presented at line 13 of Algorithm 2, note that $k > 0$. As an example, for sampling a length-3 execution in $(a + (b; c))^\star$, one may select $k = 1$ with probability $2/3$, which yields *abc* or *aaa* depending on the recursive call to $(Q^\star, 2)$ or $k = 2$, with probability $1/3$, which yields *bca*.

We did not give details on how to generate the random variable k for the parallel, sequential and loop case. Molinero showed in [20,19] that good performance can be achieved by using the so-called boustrophedonic order. For instance, in the case of the sequential composition $P = (Q; R)$, the idea is to generate a random integer x in the interval $[\![0; p_n[\![$ and to find the minimum number ℓ such that the sum of ℓ terms $q_0 r_n + q_n r_0 + q_1 r_{n-1} + q_{n-1} r_1 + q_2 r_{n-2} + \cdots$ (taken in this particular order) is greater than x. Then k is such that the last term of this sum is $q_k r_{n-k}$.

Theorem 2. *Using the boustrophedonic order, the complexity of the random generation of an execution of length n in P in terms of arithmetic operations on big integers is $O(n \cdot \min(\ln(n), h(P)))$ where $h(P)$ refers to the* height *of P i.e. its maximum number of nested operators.*

Proof. The $O(n \ln(n))$ bound follows from Theorem 11 of [20]. We obtain the other bound by refining the result from Theorem 12 from the same source. The combinatorial classes we are considering are built from the $\star$, $\boxed{\star}$, $+$ and $\textsc{Set}^{\boxed{\star}}(\cdot)$ operators without recursion, they hence fall under the scope of *iterative classes* for which Molinero proved a linear complexity in n. However the proof given in [20] does not give an explicit bound for the multiplicative constants, which actually depends on the size of the grammar and which we cannot consider constant in our context. Let $C(P, n)$ denote the cost of $\textsc{UnifExec}(P, n)$ in terms of arithmetic operations on big integers. We show that $C(P, n) \leqslant \alpha n h(P)$ by induction for some constant α to be specified later.

- The base cases have a constant cost.
- The case of the choice only incurs a constant number c of arithmetic operations in addition to the cost of the recursive call. Hence $C(Q + R, n)$ is bounded by $c + \alpha \max(C(Q, n), C(R, n)) \leqslant c + \alpha n \max(h(Q), h(R)) = c + \alpha n(h(Q + R) - 1)$ by induction. Thus, if $\alpha \geqslant c$, then $C(Q + R, n) \geqslant \alpha n h(Q + R)$.
- The parallel composition case incur a number of arithmetic operation of the form $c' \min(k, n - k)$ where k is the random variable generated using the boustrophedonic order technique. Hence $C(Q \parallel R, n)$ is bounded

by $c' \min(k, n-k) + C(Q,k) + C(R, n-k)$ and by induction by $c' \min(k, n-k) + \alpha k h(Q) + \alpha(n-k)h(R) \leqslant \alpha n h(Q \parallel R) + c' \min(k, n-k) - \alpha n$. The last term on the right is bounded by 0 if $\alpha \geqslant c'$.
- Sequential composition uses the same argument as for parallel composition.
- Finally, the loop must be handled by reasoning "globally" on the total number of unfoldings. Say the loop $Q^\star$ is unfolded r times. Then its cost $C(Q^\star, n)$ is bounded by $\sum_{i=1}^{r} c' \min(k_i, k_{i+1} + \cdots + k_r) + \sum_{i=1}^{r+1} C(Q, k_i)$. The first sum is bounded by $c'n$ and the second is bounded by induction by $\sum_{i=1}^{r+1} \alpha k_i h(Q) = \alpha n h(Q)$. Hence, reusing the bound $\alpha \geqslant c'$ we get $C(Q^\star, n) \geqslant \alpha n h(Q^\star)$ which terminates the proof.

3.3 Execution prefixes

The uniform sampler of executions described above provides one way of exploring the state space of a program, but it does not offer much flexibility. In this subsection we develop, as a complementary tool, a uniform random sampler of execution *prefixes* of a given length. This approach is not the same as using the previous algorithm until a length threshold n because this approach would not yields *uniform* prefixes. An execution prefix is a sequence of evaluation steps as in Definition 2 but unlike an execution, its resulting program P_n does not necessarily satisfy nullable(P_n). We see this algorithm as an elementary building block for statistical exploration of the state-space, enabling a variety of different exploration strategies, possibly biased towards some area of interest in the state-space of the program but in a *controlled* manner.

The idea here is to apply our previous algorithm to a new program $\mathrm{pref}(P)$ defined inductively using Table 3. Note that $\mathrm{pref}(P)$ (as well as its specification) can be implemented in linear space by using pointers to refer to the substructures of P.

Table 3: In the second column: the pref() transformation, mapping a program P to a program whose executions the prefixes of executions of P. In the third column: the combinatorial specification of the prefixes of P.

Program P	Prefix program $\mathrm{pref}(P)$	Specification of the prefixes $\langle P \rangle$
0	0	$\mathcal{E}$
a	$0 + a$	$\mathcal{E} + \mathcal{Z}$
$P \parallel Q$	$(\mathrm{pref}(P) \parallel \mathrm{pref}(Q))$	$\langle P \rangle \star \langle Q \rangle$
$P; Q$	$\mathrm{pref}(P) + (P; \mathrm{pref}(Q))$	$\langle P \rangle + S(P) \boxtimes (\langle Q \rangle \setminus \mathcal{E})$
$P + Q$	$\mathrm{pref}(P) + \mathrm{pref}(Q)$	$\langle P \rangle + (\langle Q \rangle \setminus \mathcal{E})$
$P^\star$	$P^\star; \mathrm{pref}(P)$	$\mathcal{E} + S(P^\star) \boxtimes (\langle P \rangle \setminus \mathcal{E})$

Proposition 1. *Let P be an NFJ program. The executions of the program $\mathrm{pref}(P)$ are in one-to-one correspondence with the prefixes of executions of P.*

Proof. The way the executions prefixes are defined, the transformation is direct. An atomic action a has two execution prefixes, the empty prefix and the execution consuming a. A prefix of the parallel composition $P \parallel Q$ is simply the parallel composition of one prefix of each operand. The case of the sequential composition is less trivial: a prefix of execution of $P; Q$ is either a prefix of P or a *complete* execution of P followed by a prefix of Q. Note that in order to avoid counting the complete executions of P twice in the specification, we must only consider the non-empty prefixes of Q in the second case. A prefix of a choice $P+Q$ is simply a prefix of either component. Note that in the combinatorial specification this times we always subtract $\mathcal{E}$ from $\langle Q \rangle$ to avoid double-counting the empty prefix because we know all programs have the empty prefix. Finally the case of the loop is a generalisation of the sequence: a prefix of $P^\star$ is made of any number of non-empty complete executions of P (they correspond to complete evaluation of the loop) followed by a non-empty prefix of P. A case must be added for the empty prefix in the specification.

As an example, using the notations $P_0 = P_1; (e+(f \parallel g))$, $P_1 = (P_2 \parallel (d+0))^\star$ and $P_2 = a + (b \parallel c)^\star$, the specification of the prefixes of example program P_0 is given by:

$$\begin{aligned}
\langle P_0 \rangle &= \langle P_1 \rangle + S(P_1) \boxtimes (\mathcal{Z} + \mathcal{E} + (\mathcal{Z} + \mathcal{E}) \star (\mathcal{Z} + \mathcal{E}) \setminus \mathcal{E}) \\
\langle P_1 \rangle &= \mathcal{E} + S(P_1) \boxtimes (\langle P_2 \rangle \star (\mathcal{Z} + \mathcal{E} + \mathcal{E} \setminus \mathcal{E}) \setminus \mathcal{E}) \\
\langle P_2 \rangle &= \mathcal{E} + S(P_2) \boxtimes ((\mathcal{Z} + \mathcal{E} + (\mathcal{Z} + \mathcal{E}) \star (\mathcal{Z} + \mathcal{E}) \setminus \mathcal{E}) \setminus \mathcal{E})
\end{aligned}$$

where $S(P_1)$ and $S(P_2)$ are not given but can be obtained as sub-terms of the specification $S(P_0)$ of the executions of P_0 given earlier.

Theorem 3. *To sample uniformly a prefix of length n in P we sample uniformly a full execution in $\mathrm{pref}(P)$. This has the same complexity as sampling an execution of n up to a multiplicative constant.*

The later theorem is a consequence of Proposition 1. The complexity bound is obtained by showing that the height of $\mathrm{pref}(P)$ is at most twice the height of P. Another possibility of equal complexity would be to express directly the specification of the prefixes of P without actually constructing the intermediate program. This specification is denoted $\langle P \rangle$ and is given in the third column of Table 3.

4 Experimental study

In order to assess experimentally the efficiency of our method, in this section we put into use the algorithms presented in the paper and demonstrate that they can handle systems with a significantly large state space. We generated a few NFJ programs at random using a Boltzmann random generator. In its basic form, the Boltzmann sampler would generate a high number of loops and a large number of terms of the form $P + 0$ in the programs which we believe is not realistic so we tuned it using [4] so that the number of both types of nodes

is 10% of the size of the program in average. We rely on the FLINT library (Fast Library for number theory [16]) to carry all the computations on polynomials except for the coloured product and the inversion.. The former was not provided natively by the library and the later was feasible using FLINT's primitives but slow compared to the approach based on the Newton iteration.

Note that besides the choice of the algorithms, we did not optimized our code for efficiency nor ran extensive tests on a big dataset, hence the numbers we give should be taken as a rough estimate of the performance of our algorithms. For the sake of reproducibility, the source code of our experiments is available on the companion repository[4].

4.1 Preprocessing phase

First, Table 4 gives the runtime of the preprocessing phase that computes the generating functions of all the sub-terms of a program up to a given degree n. We measured this for programs of different sizes and for different values of n. Every measure was performed 7 times and we reported the median of these 7 values. The time reported is the CPU time as measured by C's `clock` function. The state-space column indicates the number of executions of length at most n obtained by evaluating the polynomial with $z = 1$. The figure on the right displays more data and focuses on the relation between the runtime of the preprocessing (on the y axis, in seconds) and the size of the state-space (the x axis is the $\log_2$ of the number of executions). Each line corresponds to a program and each point corresponds to a different value of n for this program. Using a log-scale on both axis, this figure gives experimental "evidence" of a polynomial relation between the two. Besides the shape of the curves, the take-away here is that the preprocessing phase can be carried for systems with a state-space of size $\approx 2^{1800}$ in a time of the order of one minute.

4.2 Random generation

We then measure the runtime of the random generator of executions and execution prefixes for the same programs. Every measure was performed 100 times and we reported the median of these values. A summary of the results is available in Table 5. Interestingly, the number of executions and the numbers of execution prefixes are rather close. We have clues about the reasons behind this phenomenon which relate to the analytical properties of the generating functions of executions and prefixes. We will investigate this in the future but this is beyond the scope of this article. As the numbers show, both random sampling procedure take a few milliseconds to generate an object, even for rather large state-spaces. Here, the two state-space columns refers respectively to the number of executions and the number of prefixes of length *exactly* n.

[4] All the benchmarks were run on a standard laptop with an Intel Core i7-8665U and 32G of RAM running Ubuntu 19.10 with kernel version 5.3.0-46-generic. We are using FLINT version 2.5.2 and GMP version 6.1.2.

Table 4: On the left: runtime of the counting algorithm and size of the state-space (executions of length at most n) for programs of different sizes. On the right: plot of this runtime as a function of the $\log_2$ of the size of the state-space.

$\|P\|$	n	# exec°	runtime
100	500	$1.740825 \cdot 2^{1119}$	0.010s
100	1000	$1.073991 \cdot 2^{2235}$	0.037s
100	3000	$1.385924 \cdot 2^{6691}$	0.605s
500	500	$1.058776 \cdot 2^{1927}$	0.076s
500	1000	$1.081276 \cdot 2^{3832}$	0.462s
500	3000	$1.341591 \cdot 2^{11423}$	6.428s
1000	500	$1.473353 \cdot 2^{2330}$	0.159s
1000	1000	$1.044525 \cdot 2^{4712}$	0.874s
1000	3000	$1.092147 \cdot 2^{14181}$	13.488s
2000	100	$1.981851 \cdot 2^{410}$	0.012s
2000	200	$1.800651 \cdot 2^{926}$	0.049s
2000	500	$1.768618 \cdot 2^{2380}$	0.330s
2000	1000	$1.215440 \cdot 2^{4746}$	1.870s
5000	500	$1.607519 \cdot 2^{2923}$	0.897s
5000	1000	$1.469086 \cdot 2^{6016}$	5.434s
5000	3000	$1.226718 \cdot 2^{18116}$	75.649s

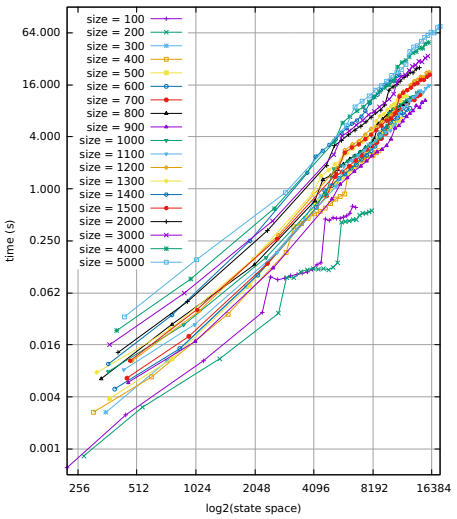

Table 5: Median and interquartile range (IQR) of the runtime of the executions and prefixes samplers for various program sizes and object lengths.

$\|P\|$	n	# exec°	UnifExec	IQR	# prefixes	UnifPrefix	IQR
100	500	$1.370 \cdot 2^{1119}$	0.129ms	4µs	$1.841 \cdot 2^{1128}$	0.147ms	3µs
100	1000	$1.690 \cdot 2^{2234}$	0.276ms	10µs	$1.124 \cdot 2^{2244}$	0.307ms	18µs
100	3000	$1.090 \cdot 2^{6691}$	1.076ms	43µs	$1.439 \cdot 2^{6700}$	1.371ms	359µs
500	500	$1.969 \cdot 2^{1926}$	0.218ms	5µs	$1.022 \cdot 2^{1997}$	0.281ms	12µs
500	1000	$1.004 \cdot 2^{3832}$	0.563ms	21µs	$1.404 \cdot 2^{3901}$	0.688ms	33µs
500	3000	$1.245 \cdot 2^{11423}$	3.718ms	203µs	$1.466 \cdot 2^{11492}$	4.005ms	274µs
1000	500	$1.420 \cdot 2^{2330}$	0.301ms	10µs	$1.556 \cdot 2^{2411}$	0.352ms	19µs
1000	1000	$1.005 \cdot 2^{4712}$	0.777ms	28µs	$1.293 \cdot 2^{4790}$	0.871ms	46µs
1000	3000	$1.051 \cdot 2^{14181}$	4.829ms	481µs	$1.127 \cdot 2^{14259}$	5.307ms	569µs
2000	500	$1.704 \cdot 2^{2380}$	0.308ms	14µs	$1.839 \cdot 2^{2484}$	0.416ms	10µs
2000	1000	$1.169 \cdot 2^{4746}$	1.021ms	51µs	$1.482 \cdot 2^{4856}$	1.225ms	86µs
2000	3000	$1.634 \cdot 2^{14120}$	7.291ms	1.2ms	$1.921 \cdot 2^{14256}$	7.245ms	238µs
5000	500	$1.589 \cdot 2^{2923}$	0.309ms	7µs	$1.933 \cdot 2^{3168}$	0.348ms	14µs
5000	1000	$1.448 \cdot 2^{6016}$	0.898ms	43µs	$1.340 \cdot 2^{6231}$	1.027ms	41µs
5000	3000	$1.208 \cdot 2^{18116}$	18.526ms	1.5ms	$1.034 \cdot 2^{18324}$	21.478ms	1.2ms

4.3 Prefix covering

This subsection presents an experimentation that highlights the importance of the uniform distribution for the purpose of state-space exploration. The setup is the following: consider a given NFJ program and randomly sample prefixes of given length n of this program using two different algorithms:

- our random sampler which is *globally* uniform among all prefixes of length n;
- a "naive" sampler that repeatedly generates one execution step uniformly among the legal steps, until we get a length n prefix. This strategy is called *locally uniform* or *isotropic*.

The question is: in average, how many random prefixes must be generated in order discover a given proportion of the possible prefixes? This question actually falls under the scope of the Coupon Collector Problem, which is treated in depth in [11]. Table 6 gives numerical answers for both exploration strategies for a random NFJ program of size 25 and for a target coverage of 20% of the possible prefixes.

Table 6: Expected number of prefixes to be sampled to discover 20% the prefixes of a random program of size 25 with either the isotropic or the uniform method.

Prefix length	1	2	3	4	5
# prefixes	11	18	30	60	128
Isotropic	2.1	4.45	11.17	35.09	$1.28 \cdot 10^{14}$
Uniform	2.1	3.18	6.57	13.26	27.69
Gain	0%	40%	70%	165%	$4.61 \cdot 10^{14}$%

Expectedly the uniform strategy is faster but what is interesting to see is that the speedup compared to the isotropic method grows extremely fast. The more the state-space grows, the more the uniform approach is unavoidable.

Unfortunately, the formula given in [11] for the isotropic case involves the costly computation of power-sets which makes it impractical to give values for larger programs and prefix length. However, these small-size results already establish a clear difference between the two methods. It would be interesting to have theoretical bounds to quantify this explosion or to investigate more efficient ways to compute these values but this falls out of the scope of this article.

References

1. Abbes, S., Mairesse, J.: Uniform generation in trace monoids. In: 40th International Symposium MFCS. pp. 63–75 (2015)
2. Arora, S., Barak, B.: Computational Complexity - A Modern Approach. Cambridge University Press (2009)
3. Basset, N., Mairesse, J., Soria, M.: Uniform sampling for networks of automata. In: 28th International Conference CONCUR (2017)
4. Bendkowski, M., Bodini, O., Dovgal, S.: Polynomial tuning of multiparametric combinatorial samplers, pp. 92–106
5. Bodini, O., Dien, M., Genitrini, A., Peschanski, F.: Entropic Uniform Sampling of Linear Extensions in Series-Parallel Posets. In: 12th International Symposium CSR. pp. 71–84 (2017)
6. Bodini, O., Dien, M., Genitrini, A., Peschanski, F.: The Ordered and Colored Products in Analytic Combinatorics: Application to the Quantitative Study of Synchronizations in Concurrent Processes. In: 14th SIAM Meeting ANALCO. pp. 16–30 (2017)
7. Bodini, O., Genitrini, A., Peschanski, F.: The Combinatorics of Non-determinism. In: IARCS Annual Conference FSTTCS. vol. 24, pp. 425–436 (2013)
8. Bodini, O., Genitrini, A., Peschanski, F.: A Quantitative Study of Pure Parallel Processes. Electronic Journal of Combinatorics **23**(1), P1.11, 39 pages (2016)
9. Bodini, O., Dien, M., Genitrini, A., Peschanski, F.: The Combinatorics of Barrier Synchronization. In: International Conference Petri Nets. pp. 386–405. Springer (2019)
10. Darrasse, A., Panagiotou, K., Roussel, O., Soria, M.: Boltzmann generation for regular languages with shuffle. In: GASCOM 2010 - Conference on random generation of combinatorial structures. Montréal, Canada (Sep 2010)
11. Flajolet, P., Gardy, D., Thimonier, L.: Birthday Paradox, Coupon Collectors, Caching Algorithms and Self-Organizing Search. D. A. Math. **39**(3), 207–229 (1992)
12. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press (2009)
13. Flajolet, P., Zimmermann, P., Cutsem, B.V.: A calculus for the random generation of combinatorial structures (1993)
14. Gaudel, M., Denise, A., Gouraud, S., Lassaigne, R., Oudinet, J., Peyronnet, S.: Coverage-biased random exploration of models. Electr. Notes Theor. Comput. Sci. **220**(1), 3–14 (2008)
15. Grosu, R., Smolka, S.A.: Monte carlo model checking. In: 11th International Conference TACAS. pp. 271–286 (2005)
16. Hart, W., Johansson, F., Pancratz, S.: FLINT: Fast Library for Number Theory (2013), version 2.5.2, http://flintlib.org
17. Koomen, C.J.: Calculus of Communicating Systems, pp. 11–26. Springer US, Boston, MA (1991)
18. Krob, D., Mairesse, J., Michos, I.: On the average parallelism in trace monoids. In: 19th Annual STACS. pp. 477–488 (2002)
19. Martínez, C., Molinero, X.: An experimental study of unranking algorithms. In: Ribeiro, C.C., Martins, S.L. (eds.) Experimental and Efficient Algorithms. pp. 326–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
20. Molinero, X.: Ordered generation of classes of combinatorial structures. Ph.D. thesis, Universitat Politècnica de Catalunya (Oct 2005)

21. Oudinet, J., Denise, A., Gaudel, M.C., Lassaigne, R., Peyronnet, S.: Uniform Monte-Carlo Model Checking. In: 14th International Conference FASE (2011)
22. Pivoteau, C., Salvy, B., Soria, M.: Algorithms for combinatorial structures: Well-founded systems and Newton iterations. Journal of Combinatorial Theory, Series A **119**, 1711–1773 (2012)