A Combinatorial Study of Async/Await Processes

Matthieu Dien, Antoine Genitrini, Frederic Peschanski

A Combinatorial Study of Async/Await Processes

Matthieu Dien, Antoine Genitrini, and Frédéric Peschanski

[1] Université de Caen – GREYC – CNRS UMR 6072 matthieu.dien@unicaen.fr
[2] Sorbonne Université, CNRS, LIP6, UMR7606.
{antoine.genitrini,frederic.peschanski}@lip6.fr

Abstract. In this paper we study families of async/await concurrent processes using techniques and tools from (enumerative) combinatorics and order theory. We consider the count of process executions as the primary measure of "complexity", which closely relates to the (in general, difficult) problem of counting linear extensions of partial orders. Interestingly, the control structures of async/await processes fall into the subclass of what we call the BIT-decomposable posets, providing an effective way to count executions in practice. We also show that async/await processes can be seen as generalizations of families of interval orders, a well-studied class of partial orders. Based on this combinatorial study, we define a variety of uniform random generation algorithms. We consider on the one side the generation of process structures, and on the other side the generation of execution paths – which is performed without requiring the explicit construction of the state-space.

Keywords: Async/await · Enumerative Combinatorics · Uniform random generation

1 Introduction

Programming concurrent systems is a notoriously difficult task with various sources of complexity, among which *asynchronism* (the lack of a global clock), *non-determinism* (the existence of multiple distinct executions/outcomes) and *state explosion* (the exponential growth of such executions) appear to stand out. Various design patterns have been proposed to simplify the task at hand. *Bulk-synchronous parallelism* (BSP) [22] is such an example of a simplified architecture for (a limited form of) parallel computing. For asynchronous systems, the principle of *async/await* concurrency has emerged as a popular abstraction, based on the concepts of *promises* and/or *futures* [18] but with dedicated syntactic constructs. Widely used programming languages offer async/await abstractions, notably Javascript (the ECMAScript 9th edition [8]), Python [20] and others[3].

The main guiding idea of our research is that the complexity of synchronization patterns for concurrent processes is closely related to the relative difficulty of counting process executions. For example, counting executions of series-parallel

[3] see https://en.wikipedia.org/wiki/Async/await

structures (such as in BSP) is easy (see e.g. [16]). At the other end of the spectrum, the lack of any obvious control structure makes counting executions akin to counting linear extensions of arbitrary posets, a $\sharp P$-complete problem (cf. [4]).

In this paper, we consider the async/await processes as combinatorial objects – considering the execution count as their fundamental "measure" – and study them with the hopeful objective of corroborating the common belief that this would be a "simpler" concurrency model. As a starting point, we develop a minimal process calculus with not much more than the basic principles of async/await. We first provide an operational interpretation of such programs in the form of computation trees. In this semantic representation, counting executions is easy: it is the number of distinct branches (thus leaves) of the trees. We then describe an alternative representation of the control structure of processes as directed acyclic graphs (DAGs). The advantage of this representation is that it is exponentially more compact than the corresponding computation tree. And we can still count executions, although the task is now more complex. As a first contribution, we show that the corresponding structure falls into the subclass of what we call the BIT-decomposable posets. Based on our previous work [4], we obtain a way to formulate the counting problem as a compact multivariate integral which can be solved by a computer algebra system. This is discussed in Section 3. In Section 4, we establish interesting links between subclasses of async/await processes and the mathematical structures known as *interval orders*. In the last part of the paper (Section 5), we build on our combinatorial investigations to experiment uniform random generation algorithms. We discuss the generation of process structures as well as the generation of execution paths.

Related work

Alternative combinatorial models of concurrency have been proposed in the literature, especially based on the *trace monoid* (see e.g. [1]). Closely related are the so-called *unfoldings* [9] which provides a compact representation of computation trees as occurrence nets (a subclass of Petri nets). However as discussed in [9] (e.g. page 29) the potentially exponential growth of the unfoldings is directly connected to the degree of synchronization exposed by the processes. The situation is similar in [3], which models synchronized automata as a product automaton of a size whose (exponential) growth is tightly connected to the number of required synchronizations. While the partial order representation we adopt is more restricted in terms of expressivity, it is less sensitive to the number of synchronizations and thus well-suited for "principled" synchronization models such as async/await. The counting of linear extensions of (unconstrained) partial orders is shown a $\sharp P$-complete problem in [6]. This is also the case for posets of height 2 or dimension 2 [7]. Polynomial algorithms exist for series-parallel posets [16]. In [4] we introduce the class of BIT-decomposable processes together with a compact representation of the counting problem as a multivariate integral formula. While this does not directly yield a counting algorithm, computer algebra systems can be used for the numerical resolution. More generic algorithms have

Listing 1.1. Async/await example in Javascript

```
function promise(arg) {
        return new Promise(resolve => {
                result = someComputation(arg);
                resolve(result)});
}

async function main() {
        // main program
        doThis();
        w1 = promise(1);
        w2 = promise(2)
        doThat();
        w3 = promise(3)
        result1 = await w1;
        use1(result1);
        result2and3 = await Promise.all([w2, w3]);
        use2and3(result2and3);
}
```

been proposed in e.g. [14] (for sparse posets, with interesting applications in artificial intelligence) and [19] (based on a fast enumeration of linear extensions, not suitable for actual counting). In [4] we describe a linear extension sampler for BIT-decomposable processes, which we experiment on async/await processes in the present paper. An alternative approach is proposed in [13] which is based on a coupling from the past (MCMC) procedure. The advantage of this approach is that it can be applied on arbitrary posets, but its running time is aleatory. Other random generation methods have been proposed in e.g. [17] but, unlike our approach, they require in one way or another the explicit construction of the state-space of processes.

Interval orders [11] have been thoroughly studied in the literature, with the notable mention of [5] which provides a thorough study in the domain of enumerative combinatorics.

2 Async/await concurrency

In this section we present a very simple process calculus whose purpose is to capture the fundamental ingredients of async/await concurrency. Listing 1.1 shows a somewhat minimal Javascript example[4]. Putting aside the classical language features (function calls, assignments, etc.), we focus on the construct related to concurrency. First, the `async` keyword enables async/await concurrency in the scope of a function body (the function `main()` in the example). In our

[4] A more complete, runnable version of the Javascript example is available online at the following address: `https://jsfiddle.net/boah97dm/`

own terminology, we will say that the body of such a function becomes a *control thread*. Such a control thread can perform three kinds of (concurrency-related) operations:

- perform basic atomic actions that have no further meaning than "something happened" as far as concurrency is concerned (in the example these are the `doThis()`, `doThat()` and `use..()` calls)
- spawn a new *promise* that will run asynchronously (this corresponds to the `promise()` calls in the example)
- await the completion of the spawned promises, based on a principle of *barrier synchronization* (in the example, the barriers are `w1`, `w2` and `w3`). Note that a control thread may wait for multiple barriers at once, in an atomic manner (using `Promise.all(...)`).

Each promise is associated to a dedicated barrier, which it has to use to *signal* its termination to the control thread (in Listing 1.1, this is the role of the `resolve()` callback). The promises can also perform atomic actions and indeed be labeled `async` to become control threads themselves. In this case, we will say that the system has a *promise depth* greater than 1 (we will come back to this important characteristic later on).

$$
\begin{array}{lll}
\text{Process } P, Q ::= 0 & : & \text{terminate the control thread} \\
\quad \mid \alpha.P & : & \text{perform action } \alpha \text{ and continue as process } P \\
\quad \mid \nu(\omega)[Q].P & : & \text{spawn promise } Q \text{ with barrier } \omega, \\
 & & \text{and continue to run } P \text{ asynchronously} \\
\quad \mid \overline{\omega} & : & \text{signal on barrier } \omega \text{ (from a promise)} \\
\quad \mid \langle \Omega \rangle.P & : & \text{await all barriers in set } \Omega, \text{ and} \\
 & & \text{after synchronization continue as } P
\end{array}
$$

$$
\begin{cases}
\text{depth}(0) = \text{depth}(\overline{\omega}) = 0 \\
\text{depth}(\alpha.P) = \text{depth}(\langle \Omega \rangle.P) = \text{depth}(P) \\
\text{depth}(\nu(\omega)[Q].P) = \max(\text{depth}(Q) + 1, \text{depth}(P))
\end{cases}
$$

Table 1. A calculus for async/await concurrency, and the definition of *promise depth*

Table 1 presents the syntax of a process calculus that captures the features listed above, and nothing much beyond that. There is also the formal definition of the promise depth introduced previously. Using the proposed syntax, our example can be abstracted as follows:

$$
\begin{aligned}
&\text{this}.\nu(\omega_1)[\text{prom}_1.\overline{\omega_1}].\nu(\omega_2)[\text{prom}_2.\overline{\omega_2}].\text{that}.\nu(\omega_3)[\text{prom}_3.\overline{\omega_3}] \\
&\quad .\langle w_1 \rangle.\text{use}_1 \ .\langle w_2, w_3 \rangle.\text{use}_{2,3}.0
\end{aligned}
$$

Once the syntactic objects under study set, we have to give them a semantic interpretation. One way of explaining the behavior of processes is to provide

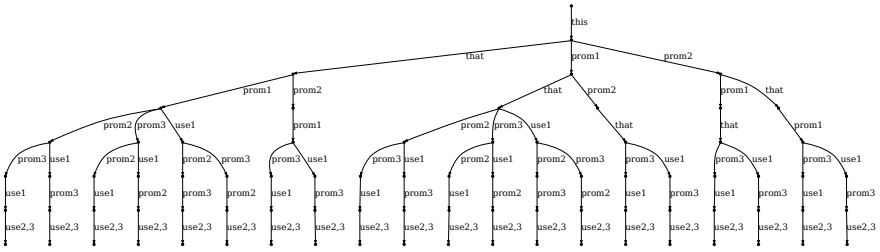

Fig. 1. A computation tree corresponding to the example of Listing 1.1

an operational semantics[5], which enables the construction of a first combinatorial object worth studying: what is colloquially called a *computation tree*. Fig. 1 depicts the operational semantics of our example as a computation tree. Each directed edge in the tree corresponds to a possible labeled transition (implicitly performing the silent steps). For example, in the initial state only the atomic action labeled "this" is possible, which leads to a state from which three distinct transitions are possible: "that", "prom_1" and "prom_2" (the other actions being blocked by await constructs), and so on. Each possible *execution path* is depicted by a distinct branch in the tree. This representation has the interesting combinatorial characteristic of relating a structural notion of a size – the number of atomic actions to perform identified with the length of each path – to a corresponding semantic notion of a size – the number of execution paths identified with the number of leaves of the tree. In the example there are 20 distinct execution paths and each path has length 7.

$$
\begin{array}{l}
[\![P]\!] = [\![P]\!]_{\circ_\top} \setminus \{\circ_\bot, \circ_\top\} \\
[\![0]\!]_x = \{x \mapsto \circ_\bot\} \\
[\![\alpha.P]\!]_x = [\![P]\!]_{\bullet_\alpha} \cup \{x \mapsto \bullet_\alpha\} \\
[\![\nu(\omega)[Q].P]\!]_{\bullet_u} = [\![\nu(\omega)[Q].P]\!]_{\circ_v} \cup \{\bullet_u \mapsto \circ_v\} \text{ with } \circ_v \text{ fresh} \\
[\![\nu(\omega)[Q].P]\!]_{\circ_v} = [\![Q]\!]_{\circ_v} \parallel_\omega [\![P]\!]_{\circ_v} \\
[\![\overline{\omega}]\!]_x = \{x \mapsto \circ_\omega\} \\
[\![\langle\Omega\rangle.P]\!]_{\bullet_u} = [\![\langle\Omega\rangle.P]\!]_{\circ_v} \cup \{\bullet_u \mapsto \circ_v\} \text{ with } \circ_v \text{ fresh} \\
[\![\langle\Omega\rangle.P]\!]_{\circ_v} = [\![P]\!]_{\circ_v} \cup \{\circ_\omega \mapsto \circ_v \mid \omega \in \Omega\}
\end{array}
$$

$$
\begin{array}{l}
\text{with } X_1 \parallel_\omega X_2 = \{x_1 \mapsto y_1 \in X_1 \mid y_1 \neq \circ_\omega\} \cup \{x_2 \mapsto y_2 \in X_2 \mid x_2 \neq \circ_\omega\} \\
\qquad\qquad\qquad\qquad \cup \{x \mapsto y \mid x \mapsto \circ_\omega \in X_1 \wedge \circ_\omega \mapsto y \in X_2\}
\end{array}
$$

Table 2. Async/await processes: partial order semantics

[5] In [4] we provide an operational semantics for a calculus of "barrier synchronization", which subsumes the async/await processes.

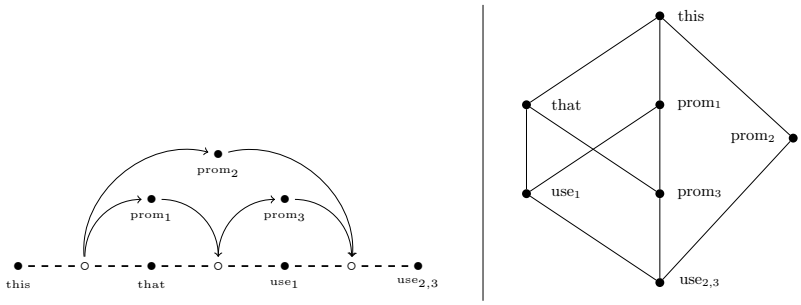

Fig. 2. The control graph (*chord process*, left) and the associated partial order (right) of the example of Listing 1.1

The main problem we are concerned with is the counting of possible execution paths of processes. With computation trees the solution is trivial since we only have to count the leaves of the tree. However, the construction of the tree itself suffers from combinatorial explosion, making this approach impractical. We thus adopt an alternative construction scheme, which is defined in Table 2. The idea is to interpret an async/await process as a directed acyclic graph (DAG) – namely its *control graph*. Fig. 2 (left) depicts the result of the construction for our example process. In the constructed graph the nodes are *events* from two complementary kinds. The *white* nodes $\circ$ encode the control-structure of the processes, i.e. when processes are forked or when they synchronize. Each *black* node, denoted by $\bullet_\alpha$, encodes the occurrence of an atomic action α. The dashed line at the bottom corresponds to the control thread, and the *chords* above correspond to promises. This DAG has several good properties. First of all, its size is linear in the syntactic size of processes, hence there is no "explosion" involved at this step. Moreover, (intransitive) DAGs are tightly related to partial orders in that they correspond to their *transitive reduction*. In the computation tree of Fig. 1 only the labels of the atomic actions are considered. We can perform a similar abstraction on the control graph be removing the white nodes while maintaining the relations among the black ones.

Definition 1 (Partial orders of async/await processes).
Let P be an async/await process. We define: $\mathcal{PO}(P) = \{\alpha > \beta \mid \bullet_\alpha \mapsto \circ_u, \circ_u \mapsto \bullet_\beta \in [\![P]\!]\}^{\textit{refl-trans}}$ with, for a binary relation R, $R^{\textit{refl-trans}} = \bigcup_{n\geq 0} R^n$.

On the right of Fig. 2 is depicted the resulting poset using the most common representation as a Hasse diagram. There is an important connection between computation trees and such partial order semantics.

Theorem 1. *The number of (non-silent) transitions of an async/await process P, hence the number of leaves of its computation tree, corresponds to the number Ψ_P of linear extensions of $\mathcal{PO}(P)$*

Proof. This is a corollary of [4, Proposition 2.1], in which we consider a class of concurrent systems more general than that of async/await processes. □

3 Partial order decomposition and the counting problem

We now investigate the problem of counting the execution paths of an async/await process P, based on the DAG representation $[\![P]\!]$ or, alternatively, its abstraction $\mathcal{PO}(P)$ as discussed in the previous section. The problem boils down to the counting of linear extensions in families of posets closely related to async/await processes[6].

Series $P \odot Q = (X_P \uplus X_Q, <_P \cup <_Q \cup (X_P \times X_Q))$	$\Psi_{P \odot Q} = \Psi_P \cdot \Psi_Q$
Parallel $P \parallel Q = (X_P \uplus X_Q, <_P \cup <_Q)$	$\Psi_{P \parallel Q} = \binom{\lvert P\rvert+\lvert Q\rvert}{\lvert P\rvert} \cdot \Psi_P \cdot \Psi_Q$

with $P = (X_P, <_P)$ and $Q = (X_Q, <_Q)$

Table 3. Series-parallel constructions and associated counting formulas (cf. [16]).

To our knowledge, there are very few (non-trivial) poset subclasses for which the counting problem can be said to be "easy". One remarkable example is that of *series-parallel* posets with dedicated and simple counting formulas, as described in Table 3. Async/await processes are not, in general, decomposable with only series and parallel operators. However, some of them *are* and, most importantly, one can often find series-parallel substructures in larger processes. This means that it is sometimes possible to use the series-parallel counting formulas, which we take advantage of in Section 5.

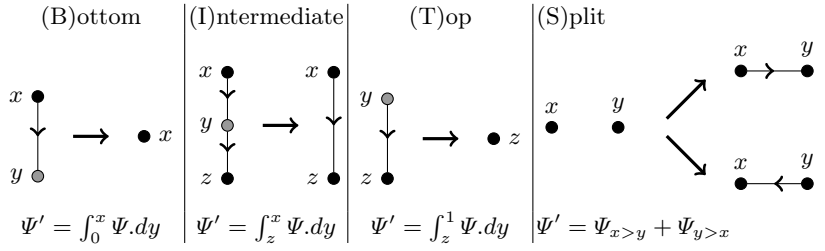

Fig. 3. BITS-decomposition and associated counting formula (from [4])

In [4] we define an alternative decomposing scheme for arbitrary partial orders. This so-called BITS-decomposition, summarized in Fig. 3, consists in applying "elimination rules" on the transitive reduction of a partial order, or equivalently on any intransitive DAG. The B-rule allows to remove a bottom node y, hence with in-degree 1 and out-degree 0. The T-rule is the complement for top

[6] We denote by $X_1 \uplus X_2$ the *disjoint sum* of the two sets X_1 and X_2.

nodes. The I-rule eliminates internal nodes with in and out degrees 1. Finally the S-rule consists in replacing two nodes x and y that are incomparable in the poset (they could both have parents or children in the DAG, which is not represented here), by two cases: first x is larger than y or the reverse y is larger than x. Obviously this rule induces a *split* requiring to consider then two distinct sub-orders. Most importantly, a symbolic formula Ψ for the linear extensions count (of the induced poset) can be constructed along the decomposition. Moreover, if one manages to only use the BIT-rules for decomposing a poset – which is then qualified as BIT-decomposable – then the formula we obtain is of a linear size. This does not mean that the counting problem itself becomes easy (in fact we conjecture it remains $\sharp P$-complete), however we get: (1) a concise way to formulate it, and (2) an effective way of computing the result using a computer algebra system.

One of the main result of the present paper follows.

Theorem 2. *The control graph $[\![P]\!]$ of an async/await processes P is BIT-decomposable.*

Proof (Proof sketch). The construction ensures that the black nodes have all in-degree and out-degree exactly one. Indeed, only white nodes can have in-degree or out-degree > 1 representing join or spawn events. The I-rule of the construction is thus powerful enough to remove the black nodes. Now, we consider the promises with maximal depth (i.e. promises not making further promises). Since such a promise cannot spawn a process, it has no white node except its fork and join events. Since we can remove its black nodes the promise itself can be removed, which means the out-degree of its fork node is reduced by one, and so is the in-degree of its join node. Hence, all promises of maximal depth can be removed by the BIT-rules. Once removed, their parent promises become of maximal depth, and by a simple inductive argument we conclude that the whole structure is decomposable. □

Here is an example of the counting formula generated thanks to the decomposition of the control graph of Fig. 2 (white nodes are labeled by w_1, w_2 and w_3 from left to right):

$$\int_0^1 \int_0^{w_1} \int_0^{w_2} \int_0^{w_3} \int_{w_1}^1 \int_{w_3}^{w_1} \int_{w_3}^{w_2} \int_{w_2}^{w_1} \int_{w_2}^{w_1} \int_{w_3}^{w_2} 1 \; d\mathrm{use}_1 d\mathrm{that} d\mathrm{prom}_1 d\mathrm{prom}_3 d\mathrm{prom}_2 d\mathrm{this} d\mathrm{use}_{2,3} dw_3 dw_2 dw_1.$$

Once evaluated, this multivariate integral formula produces the value 24, which corresponds to the number of possible executions paths in the control graph. Note that this is more than the 20 possible execution branches of the computation tree of Fig. 1, since the white nodes are also taken into account and not just the atomic actions. If we consider the partial order $\mathcal{PO}(P)$ with all white nodes abstracted away, then BIT-decomposability is not guaranteed anymore. In the next section this is discussed more thoroughly but we can still consider the poset of (the right of) Fig. 2 as an illustration. The only node with input/output arity one is the

node labeled "prom_2" and the other nodes have an arity > 1 even if "prom_2" is deleted. Hence, this poset is *not* BIT-decomposable. However, luckily, it can be decomposed in series-parallel, as follows: this $\odot$ (((that $\parallel$ prom_1) $\odot$ (use_1 $\parallel$ prom_3)) $\parallel$ prom_2) $\odot$ $\mathrm{use}_{2,3}$.

We can compute the number of linear extensions according to Table 3, which, schematically, gives:

$$1 \cdot ((\binom{2}{1} \cdot 1 \cdot 1) \cdot (\binom{2}{1} \cdot 1 \cdot 1) \parallel 1) \cdot 1 = (2 \cdot 2) \parallel 1 = \binom{4+1}{4} \cdot 4 \cdot 1 = 5 \cdot 4 = 20.$$

This is of course the number of leaves of the computation tree of Fig. 1.

4 Chord processes, interval orders and related families

We now plunge more deeply into combinatorics questions. Our objective is to characterize relevant subclasses of async/await processes in a constructive way, following the principles of the *symbolic method* [12, part A]. The basic idea is to use generating functions to enumerate, symbolically, the constructed objects of a given size in the considered combinatorial class, and to derive an inductive equation (or less constrained, a functional equation) satisfied by such function. Most importantly, our ultimate goal is to relate such process subclasses to corresponding classes of partial orders. This imposes that we somewhat restrict the possibilities of constructing processes. In this section, we adopt the following constraints. First, we require that a promise spawned by a process performs exactly one atomic action at start-up time, which means that atomic actions and promises are somehow identified. Then, the promise may spawn one of several promises of greater depth, but ultimately it has to signal on its dedicated barrier. Moreover, we will make sure that the number of white nodes is minimized in the control graphs. Because of its proximity to what is called a chord diagram elsewhere [21], the class we study in this section will be named *chord processes*.

Our first subclass of interest, named $\mathcal{S}_1$, considers chord processes with two further restrictions: (1) the *depth* of the process is one, and (2) there is no redundant promise. The first constraint means that there is exactly one control thread, thus promises cannot spawn further promises. For the second constraint, we consider two promises to be redundant if they are spawned and synchronized at the same time (i.e. they have the same origin and destination white nodes in the control graph). The class $\mathcal{S}_1$ corresponds to so-called *non-redundant* chord processes.

An example of a control graph of a process in the class $\mathcal{S}_1$ is depicted in Fig. 4 (left). We consider the size of such a process to be the number of black nodes in the control graph, hence here the size is 8. The abstraction from white nodes is essential to properly capture the order-theoretic nature of the construction. Note however that the white nodes are still part of the construction, only they do not participate in the size of the objects.

In order to explain the construction of the class $\mathcal{S}_1$ properly, we need to consider a slightly larger class, named $\mathcal{S}_1^+$, whose multivariate generating function

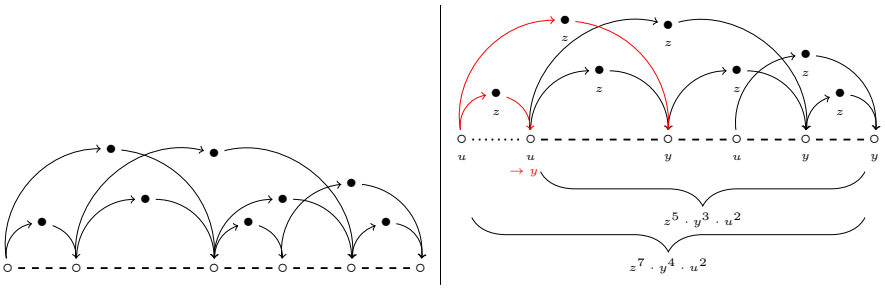

Fig. 4. (left) A process of class $\mathcal{S}_1$; (right) A process with monomial $z^5y^3u^2$, preceded by a new node u that spawns 2 promises (the new monomial is $z^7y^4u^2$).

can be written as follows:

$$S_1^+(z, y, u) = \sum_{n,k,\ell \geq 0} s_{n,k,\ell}\ z^n\ y^k\ u^\ell$$

We need no less than three parameters in this definition. First, the main parameter is z which represents the size of the objects[7]. The variable y is used to count the white nodes that can be simultaneous async and await events, while u counts the remaining white nodes that are "only async". Thus, in the definition above, the coefficient $s_{n,k,\ell}$ corresponds to the number of processes in the $\mathcal{S}_1^+$ class with n black nodes, k async and await white nodes, and ℓ async only nodes.

In the right part of Fig. 4, we illustrate how a larger process can be constructed from a smaller one while preserving the constraints of the considered combinatorial class $\mathcal{S}_1^+$. The process delimited by the "interior" brace is of size 5 (its number of black nodes), with 3+2 white nodes. The "exterior" brace delimits a larger process consisting in prepending a "async only" (u) white node on the control thread, here with two non-redundant spawned promises[8]. This process increases the size by 2 because each promise performs an action, and we can see that a previously "async only" (u) white node now serves as an await and thus becomes a y node. Hence the larger process is characterized by the monomial $z^7 \cdot y^4 \cdot u^2$.

Summarizing all the possibilities of such incremental constructions, we now define more formally the combinatorial class $\mathcal{S}_1^+$ as follows.

Definition 2. *The generating function for the combinatorial class $\mathcal{S}_1^+$ is such that:*

$$S_1^+(z, y, u) = u + u\left(S_1^+(z, y + y \cdot z, u + y \cdot z) - S_1^+(z, y, u)\right). \tag{1}$$

The smallest process satisfying the equation is of size 0, consisting in just a single white node u (the first summand of the right-hand side of the equation).

[7] We remind the reader that generating function are formal *power series*, counting "things" through polynomial degrees.

[8] To be non-redundant from the same origin, the promises must have distinct destination white nodes.

The basic principle to obtain a larger processes S' from a smaller process S is given by the second summand. This consists in adding a new white node of kind u at the beginning of the control thread (as illustrated on the right of Fig. 4), which corresponds to the u in factor in the equation. We then have to account for all the possibilities to connect this new "async" node with the rest of the process S with new promises. We consider each white node of S in turn. If it is a y node then either it stays the same, or it receives a new promise originating from the new u, hence becoming $y \cdot z$, the new z corresponding to the action of the spawned promise. If it is a u then either it is left untouched or it becomes a $y \cdot z$. In the construction, we force the new white node u to spawn at least one promise to the previous process, which is why we subtract the term $S_1^+(z, y, u)$ in the equation above, where no promise has been spawned.

Proposition 1. *Definition 2 is a sound and non-ambiguous construction.*

Proof. For the soundness part we need to prove that $\mathcal{S}_1^+$ effectively describes a combinatorial class. This means that for a given size n (the number of black nodes), there is a finite number of structures satisfying the equation of Definition 2. First, we identify the number of promises and the number of black nodes, thus there are exactly n promises for size n. The equation also enforces that each u or y node spawns at least one promise (except for the rightmost white node), so there are at most $n + 1$ such white nodes. Hence, for each such promise it remains in the worst case $n + 1$ white nodes as destination so a simple upper bound in the number of possibilities is $(n+1)^n$. Thus, the number of admissible structures of size n is indeed finite.

The second important characteristics of a functional equation enumerating a combinatorial class is that it does not construct several times the same object (i.e. it is non-ambiguous). This is a fundamental characteristics because we use this functional equation to derive the number of objects of a given size. This property can be demonstrated by structural induction. For the base case, we consider the fact that there is a single minimal structure, namely $z^0 \cdot y^0 \cdot u^1$. And for the inductive case, if we suppose that a structure S has been constructed non-ambiguously then all the possible "one-step" larger structures S' are also obtained non-ambiguously through the equation of Definition 2. □

The process class we are looking for is not directly $\mathcal{S}_1^+$ but a slightly restricted variant in which all the white nodes must be of the y kind, with the exception of the leftmost one of kind u. This accounts for our initial constraint that white nodes are minimized. Thus, the subsequence we consider is $(\cup_{k\geq 0} s_{n,k,1})_{n\in\mathbb{N}}$ in which only one parameter remains: the size n. We regroup all processes of size n regardless their number of white nodes. From this we obtain one of the main technical results of this paper.

Theorem 3. *The univariate generating function enumerating the processes from $\mathcal{S}_1$ by their number of atomic actions is given by the explicit equation:*

$$\mathcal{S}_1(z) = \frac{\partial S_1^+}{\partial u}(z, 1, 0) = 1 + \sum_{k\geq 1} \prod_{i=1}^{k} \left(1 - \frac{1}{(1+z)^i}\right).$$

The intuition here follows from Definition 2 taking into account the restriction to S_1 (having one white node), and abstracting away from the counting of the other white nodes. This explains the derivative on the left-hand side of the equation followed by the partial assignment $y \leftarrow 1; u \leftarrow 0$.

Proof. We are starting from Definition 2, but the exact enumeration for $\mathcal{S}_1$, where only the actions are counted and the white nodes do not count anymore is such that in the processes enumerated by $S_1^+(z, y, u)$ only the last added white node u is not an await node (all other white nodes are thus been marked by y) and we do not care about the number of these white nodes. So the generating function for $\mathcal{S}_1$ is given by $\dfrac{\partial S_1^+}{\partial u}(z, 1, 0)$. In fact, since we are interested in the monomials $\gamma\ z^n\ y^k\ u$, once differentiated according to u, they are not depending on u anymore, and then evaluating at $u = 0$ erases monomials where there still remains the u variable. Finally, letting $y = 1$, regroups together all monomials $\gamma' z^n$. Thus, with a partial differentiation in u:

$$\frac{\partial S_1^+}{\partial u}(z, y, u) = \frac{1}{(1+u)^2}\left(1 + S_1^+(z, y\cdot(1+z), u + y\cdot z)\right) + \frac{u}{1+u}\frac{\partial S_1^+}{\partial u}(z, y\cdot(1+z), u + y\cdot z).$$

Then by evaluating y at 1 and u at 0, it remains

$$\frac{\partial S_1^+}{\partial u}(z, 1, 0) = 1 + S_1^+(z, 1+z, z).$$

Before going on, let us simplify Equation (1) so that:

$$S_1^+(z, y, u) = \frac{u}{1+u}\left(1 + S_1^+(z, y + y\cdot z, u + y\cdot z)\right).$$

Now by injecting the latter equation we obtain

$$\frac{\partial S_1^+}{\partial u}(z, 1, 0) = 1 + \frac{z}{1+z}\left(1 + S_1^+(z, (1+z)^2, z + (1+z)z)\right).$$

By iterating this substitution we get:

$$\begin{aligned}\frac{\partial S_1^+}{\partial u}(z, 1, 0) = 1 &+ \sum_{k=0}^{n}\prod_{i=0}^{k} z\frac{(1+z)^0 + \cdots + (1+z)^i}{1 + (1+z)^0 z + \cdots + (1+z)^i z} \\ &+ \prod_{i=0}^{n} z\frac{(1+z)^0 + \cdots + (1+z)^i}{1 + (1+z)^0 z + \cdots + (1+z)^i z} \\ &\cdot S_1(z, (1+z)^{n+2}, (1+z)^0 z + \cdots + (1+z)^{n+1} z).\end{aligned}$$

Letting n tending to infinity we finally get

$$\begin{aligned}\frac{\partial S_1^+}{\partial u}(z, 1, 0) &= 1 + \sum_{k\geq 0}\prod_{i=0}^{k} z\frac{(1+z)^0 + \cdots + (1+z)^i}{1 + (1+z)^0 z + \cdots + (1+z)^i z} = 1 + \sum_{k\geq 0}\prod_{i=0}^{k} z\frac{\frac{1-(1+z)^{i+1}}{1-(1+z)}}{1 + z\frac{1-(1+z)^{i+1}}{1-(1+z)}} \\ &= 1 + \sum_{k\geq 1}\prod_{i=1}^{k}\frac{(1+z)^i - 1}{(1+z)^i} = 1 + \sum_{k\geq 1}\prod_{i=1}^{k}\left(1 - \frac{1}{(1+z)^i}\right).\end{aligned}$$

And the stated results are proved. □

Based on this theorem, we can compute the counting sequence of $\mathcal{S}_1$ processes. The first numbers from size 1 to 14 are as follows:

$$1, 1, 2, 5, 16, 61, 271, 1372, 7795, 49093, 339386, 2554596, 20794982, 182010945.$$

We think that this is a remarkable result since the sequence is in fact already known as `OEIS A138265`[9], which is the enumeration of rigid (unlabeled) interval orders [15]. Indeed, this class of partial order is characterized by the same functional equation, which establishes a one-to-one correspondence between the thoroughly studied class of interval orders [11] and the async/await processes. The interval orders are counted by the sequence of *Fishburn numbers* stored in `OEIS A022493`. The numbers of interval orders of sizes 1 to 14 are

1, 2, 5, 15, 53, 217, 1014, 5335, 31240, 201608, 1422074, 10886503, 89903100, 796713190.

It is interesting to see if we can define a class of async/await processes that exactly matches the interval orders. We of course consider the class $\mathcal{S}_1^+$ as a starting point, since it corresponds to a restriction of interval orders. In fact, what distinguishes `OEIS A138265` from `OEIS A022493` is precisely the notion of redundant promise we introduced previously. We thus consider the subclass $\mathcal{S}_2$ of the async/await processes, which corresponds to the class $\mathcal{S}_1$ but allowing redundant promises.

Theorem 4. *We consider the class $\mathcal{S}_2$ defined by the following equations:*

$$S_2(z) = \frac{\partial S_2^+}{\partial u}(z,1,0) \text{ with } S_2^+(z,y,u) = S_1^+\left(\frac{z}{1-z}, y, u\right).$$

The processes in $\mathcal{S}_2$ are in one-to-one correspondence with interval orders.

Proof. Similarly to the previous results, we introduce an auxiliary class of processes, namely $\mathcal{S}_2^+$, defined from $\mathcal{S}_1^+$ (of Definition 2) but in which we allow to substitute each promise by a finite sequence of redundant promises. Thus the z in the definition becomes $z/(1-z)$ which is the closed formula for non-empty sequences (of z's). Now, from Theorem 3 and applying the substitution we obtain the equation:

$$\frac{\partial S_2^+}{\partial u}(z,1,0) = 1 + \sum_{k\geq 1}\prod_{i=1}^{k}\left(1-(1-z)^i\right).$$

This equation exactly matches the generating function proposed in [5] to count the (unlabeled) interval orders. □

As an interesting corollary, we remark that the bivariate generating functions $y\dfrac{\partial S_1^+}{\partial u}(z,y,0)$ and $y\dfrac{\partial S_2^+}{\partial u}(z,y,0)$ can be calculated easily from the previous

[9] Throughout this paper, a reference `OEIS A···` points to an entry of Sloane's Online Encyclopedia of Integer Sequences `www.oeis.org`.

theorems. These characterize the distributions of the number of white nodes in processes of a given size, which correspond to sequences already studied in the context of interval orders (respectively in `OEIS A137252` and `OEIS A137251`). The concerned parameter of interval orders is called the *magnitude* [10]. There is a simple interpretation of the magnitude in terms of concurrency: this is the number of white nodes in the control thread of a chord process (respectively without or with redundant promises).

Once the connection with existing mathematical structures established, it is interesting to look for possible variations inspired by concurrency aspects. For now we considered chord processes of depth 1 and identified them with interval orders. It seems thus quite natural to investigate the process structures of depth > 1. The basic technical principle at work is the possibility to substitute subprocesses within processes through substitutions in the equations for the associated generating functions. For example, to construct $\mathcal{S}_2^+$ from $\mathcal{S}_1^+$ we substituted a promise by a sequence of promises. Accordingly, it seems possible to substitute a promise by a whole chord process. This way, from a subprocess of depth n we can construct a process of depth $n+1$. In order to obtain a sound and non-ambiguous generalization of interval orders (of depth > 1), we must ensure that the elementary subprocesses are proper chord processes (hence "simple" interval orders). This leads to the following definition.

Definition 3. *The class $\mathcal{S}_3$ of generalized interval orders is defined by the equation:*

$$S_3(z) = \frac{\partial S_3^+}{\partial u}(z,1,0)$$
$$\text{with } S_3^+(z,y,u) = \frac{u}{1+u}\left(1 + S_3^+\left(z, \frac{y}{1-z(1+S_3(z))}, u + y\frac{z(1+S_3(z))}{1-z(1+S_3(z))}\right)\right).$$

If compared to the previous equations, this definition is recursive so that whole sub-processes can be substituted.

Despite this extra complexity, it is still quite possible to enumerate efficiently the terms of the counting sequence. The number of processes in class $\mathcal{S}_3$ from size 1 to 14 are

1, 3, 12, 56, 289, 1606, 9471, 58790, 382496, 2604284, 18564013, 138808595, 1092001289, 9070517772.

This sequence is not present yet in OEIS and will thus be submitted for contribution.

5 Uniform random generation: experimental study

In this section we present an experimentation of combinatorial algorithms directly connected to our study. Our objective is more to highlight the kinds of problem that can be solved in practice based on our combinatorial study, rather than a detailed description of the algorithms themselves. However, the whole source code of the experiment is available online in a complement repository[10] with detailed instructions.

[10] cf. `https://gitlab.com/ParComb/async-await-randgen`

Generating structures We investigate the generation of process structures using three complementary ways. First, there is the systematic enumeration of the structures by size. Since for each finite size we know that there is a finite number of possible structures, the second interesting way of generating a structure is through what is called *unranking*: construct the k-th structure of a given size n. Last but not least the generation of structures uniformly at random represents an interesting way to validate experimentally conjectures about said structures. In our case, this provides us a way to compare algorithms based on different techniques, without having too much of bias in the comparison.

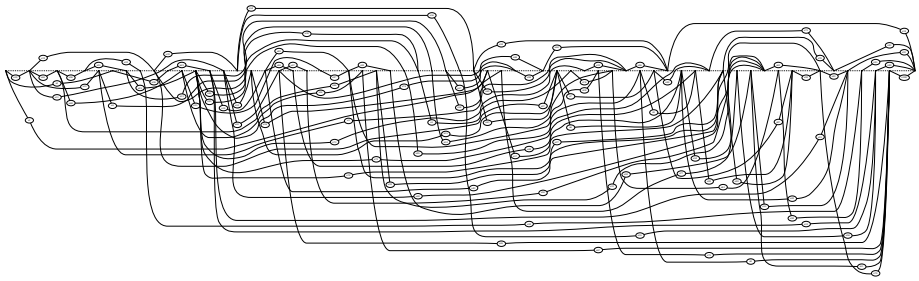

Fig. 5. The chord process corresponding to the unranking of ascent sequence of size $n = 100$ and rank $k = S_2[100]/2 - 1$

For all these needs, our starting point is [5] in which a constructive bijection between interval orders and *ascent sequences* is proposed. Quoting `OEIS A022493`:

> An ascent sequence is a sequence $[d(1), d(2), \ldots, d(n)]$ where $d(1) = 0$, $d(k) \geq 0$, and $d(k) \leq 1 + asc([d(1), d(2), \ldots, d(k-1)])$ where $asc(.)$ counts the ascents of its argument.

The enumeration of ascent sequences is easy, and moreover counting the number of such sequences of a given size n can be performed in polynomial time (in $O(n^3)$ arithmetic operations). This gives us first a quick way for unranking a sequence. Ascent sequences of a given length n can be recursively decomposed so the classical *recursive method* of [2] can be used to design a relatively efficient random sampler (in $O(n^2)$, once the complete counting of the number of structures until size n has been performed). The output of the sampler is an ascent sequence, i.e. a list of numbers, which we have to convert to an interval order exploiting the bijection of [5]. In a further step we can use the inverse of the construction of Table 2 to obtain a corresponding control graph, from which a process expression can be easily obtained. This way, we obtain a uniform random sampler for chord processes. As an illustration, in Fig. 5 we give an example of a generated chord process of size 100 from an unranked ascent sequence. More precisely, the sequence has rank $S_2[100]/2 - 1$ (where $S_2[100]$ is the number of sequences of size 100).

Uniform random generation of execution paths Based on the algorithm described in [4], we now experiment the uniform random generation of execution paths of chord processes. The main interest of this algorithm is that it does not require the explicit construction of the state space of processes.

Size	**Rank**	**Nb. paths**	**Counting time** (s)	**Random Gen.** (avg. s)
10	81694	$\approx 8.0\ e^{5}$	$4.1\ e^{-4}$ s	$3.1\ e^{-3}$ s
15	7122308736	$\approx 1.9\ e^{6}$	$2.2\ e^{-3}$ s	$4.5\ e^{-3}$ s
20	230090562434702	$\approx 1.3\ e^{13}$	$4.1\ e^{-2}$ s	$2.3\ e^{-2}$ s
25	113615274237648394333	$\approx 2.2\ e^{17}$	1.5 s	$6.3\ e^{-1}$ s
30	31410947907369433055682329 8	$\approx 2.2\ e^{24}$	3.4 s	1.3 s

Table 4. Uniform random generation of execution paths.

In Table 4 we provide the results of a preliminary benchmark of our random sampler for execution paths. The computer used for the experiments runs on GNU/Linux (Ubuntu 20.04), with a Intel Core i7-6700 CPU cadenced at 3.40GHz and 8Go of RAM. The input of the algorithm are random chord processes generated as explained previously. For each sampled process (described by its size and rank), we give the result of the counting procedure and the associated timing. And finally we generate 10 execution paths and provide the average generation time. While the implementation of the algorithm is at a very early stage of development, we think that the timing results show promising figures. Indeed, it is possible to generate execution paths uniformly at random in processes in a reasonable time, in the order of a few seconds in a size 30 process with quite a large state-space.

6 Conclusion and future work

The interpretation of concurrent systems as combinatorial objects is, we think, quite an insightful perspective. Our measure of the "complexity" of concurrent systems is that of counting execution paths. From this point of view, we show that thanks to BIT-decomposability the counting problem is in a way "simpler" for async/await processes than for arbitrary ones (in [4]). Complementary, interval orders can be seen as basic generators for async/await control paths, as suggested by our combinatorial investigation. While it is arguably a kind of a stretch, this correlates the practical experience that async/concurrency is a "simpler" form of concurrency, easier to deal with than less constrained forms. Taking this perspective upside-down, async/await processes can be seen as a generalization of interval orders, and are thus worth studying from a purely combinatorial point of view. Our section on experimenting with uniform random generation algorithms is mostly proposed as a proof of concept. We argue that there is an interest in developing analysis methods based on such building blocks. A strong argument in favor is that the algorithms can be applied directly on (the control graph of) processes without having to unfold the state-space.

References

1. Abbes, S., Mairesse, J.: Uniform generation in trace monoids. In: MFCS 2015. LNCS, vol. 9234, pp. 63–75. Springer (2015)
2. Albert, N., S., W.H.: Combinatorial Algorithms. Academic Press, New York (1978)
3. Basset, N., Mairesse, J., Soria, M.: Uniform sampling for networks of automata. In: Meyer, R., Nestmann, U. (eds.) CONCUR 2017, September 5-8, 2017, Berlin, Germany. LIPIcs, vol. 85, pp. 36:1–36:16 (2017)
4. Bodini, O., Dien, M., Genitrini, A., Peschanski, F.: Quantitative and algorithmic aspects of barrier synchronization in concurrency. Discret. Math. Theor. Comput. Sci. **22**(3) (2021)
5. Bousquet-Mélou, M., Claesson, A., Dukes, M., Kitaev, S.: (2+2)-free posets, ascent sequences and pattern avoiding permutations. J. Comb. Theory, Ser. A **117**(7), 884–909 (2010)
6. Brightwell, G., Winkler, P.: Counting linear extensions is ♯p-complete. In: Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing. p. 175–181. STOC '91, Association for Computing Machinery, New York, NY, USA (1991)
7. Dittmer, S., Pak, I.: Counting linear extensions of restricted posets. Electron. J. Comb. **27**(4), P4.48 (2020)
8. ECMA international: ECMAScript Language Specification, 9th edition edn. (2018)
9. Esparza, J., Heljanko, K.: Unfoldings. Springer (2008)
10. Fishburn, P.C.: Interval lengths for interval orders: A minimization problem. Discrete Mathematics **47**, 63–82 (1983)
11. Fishburn, P.C.: Interval graphs and interval orders. Discrete Mathematics **55**(2), 135–149 (1985)
12. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press (2009)
13. Huber, M.: Fast perfect sampling from linear extensions. Discret. Math. **306**(4), 420–428 (2006)
14. Kangas, K., Hankala, T., Niinimäki, T.M., Koivisto, M.: Counting linear extensions of sparse posets. In: IJCAI 2016. IJCAI/AAAI Press (2016)
15. Khamis, S.M.: Exact counting of unlabeled rigid interval posets regarding or disregarding height. Order **29**(3), 443–461 (2012)
16. Mörhing, R.H.: Algorithms and Order (Edited by Ivan Rival), chap. Computationally tractable classes of ordered sets, p. 127. Kluwer Academic Publishers (1987)
17. Oudinet, J., Denise, A., Gaudel, M.C., Lassaigne, R., Peyronnet, S.: Uniform monte-carlo model checking. In: International Conference on Fundamental Approaches to Software Engineering. pp. 127–140. Springer (2011)
18. Prasad, K., Patil, A., Miller, H.: Programming Models for Distributed Computing, chap. Futures and Promises (2017)
19. Pruesse, G., Ruskey, F.: Generating linear extensions fast. SIAM J. Comput. **23**(2), 373–386 (1994)
20. Selivanov, Y.: PEP 492 – Coroutines with async and await syntax. Python Org. (2015)
21. Stoimenow, A.: Enumeration of chord diagrams and an upper bound for vassiliev invariants. Journal of Knot Theory and Its Ramifications **07**(01), 93–114 (1998)
22. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (aug 1990)