Unfolding-based Partial Order Reduction*

César Rodríguez[1], Marcelo Sousa[2], Subodh Sharma[3], and Daniel Kroening[4]

1 Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, France
2,4 Department of Computer Science, University of Oxford, UK
3 Indian Institute of Technology Delhi, India

Abstract

Partial order reduction (POR) and net unfoldings are two alternative methods to tackle state-space explosion caused by concurrency. In this paper, we propose the combination of both approaches in an effort to combine their strengths. We first define, for an abstract execution model, unfolding semantics parameterized over an arbitrary independence relation. Based on it, our main contribution is a novel stateless POR algorithm that explores at most one execution per Mazurkiewicz trace, and in general, can explore exponentially fewer, thus achieving a form of *super-optimality*. Furthermore, our unfolding-based POR copes with non-terminating executions and incorporates state-caching. On benchmarks with busy-waits, among others, our experiments show a dramatic reduction in the number of executions when compared to a state-of-the-art DPOR.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Partial-order reduction, unfoldings, concurrency, model checking

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Efficient exploration of the state space of a concurrent system is a fundamental problem in automated verification. Concurrent actions often interleave in intractably many ways, quickly populating the state space with many equivalent but unequal states. Existing approaches to address this problem can essentially be classified as either partial-order reduction techniques (PORs) or unfolding methods.

POR methods [18, 7, 6, 8, 20, 19, 2, 1] conceptually exploit the fact that executing certain transitions can be postponed owing to their result being independent of the execution sequence taken in their stead. They execute a provably-sufficient subset of transitions enabled at every state, computed either statically [18, 7] or dynamically [6, 2]. The latter methods, referred as dynamic PORs (DPORs), are often *stateless* (i.e., they only store one execution in memory) and constitute the most promising algorithms of the family. By contrast, unfolding approaches [14, 5, 3, 10] model execution by partial orders, bound together by a conflict relation. They construct finite, complete prefixes by a saturation procedure, and cope with non-terminating executions using cutoff events [5, 3].

While a POR can employ arbitrarily sophisticated decision procedures to choose a sufficient subset of transitions to fire, in most cases [7, 6, 8, 20, 19, 2, 1] the *commutativity of transitions* is the enabling mechanism underlying the chosen procedure. Commutativity, or independence, is thus a mechanism and not necessarily an irreplaceable component of a

* This research was supported by ERC project 280053 (CPROVER).


© César Rodríguez, Marcelo Sousa, Subodh Sharma, Daniel Kroening;
licensed under Creative Commons License CC-BY



POR [18, 9].[1] PORs that exploit such commutativity conceptually establish an equivalence relation on the sequential executions of the system and explore at least one representative of each class, thus discarding equivalent executions. In this work we restrict our attention to exclusively PORs that exploit commutativity.

Despite impressive advances in the field, both unfoldings and PORs have shortcomings. We now give six of them. Current unfolding algorithms (1) need to solve an NP-complete problem when adding events to the unfolding [14], which seriously limits the performance of existing unfolders as the structure grows. They are also (2) inherently *stateful*, i.e., they cannot selectively discard visited events from memory, quickly running out of it. PORs, on the other hand, explore Mazurkiewicz traces [13], which (3) often outnumber the events in the corresponding unfolding by an exponential factor (e.g., Fig. 2 (d) gives an unfolding with $2n$ events and $\mathcal{O}(2^n)$ traces). Furthermore, DPORs often (4) explore the same states repeatedly [19], and combining them with stateful search, although achieved for non-optimal DPOR [19, 20], is difficult because of the dynamic nature of DPOR [20]. More on this in Example 1. The same holds when extending DPORs to (5) cope with non-terminating executions (note that a solution to (4) does not necessarily solve (5)). Lastly, (6) existing stateless PORs do not exploit additional available memory for any other purpose.

Either readily available solutions or promising directions to address these six problems can be found in, respectively, the opposite approach. PORs inexpensively add events to the current execution, contrary to unfoldings (1). They easily discard events from memory when backtracking, which addresses (2). On the other hand, while PORs explore Mazurkiewicz traces (*maximal configurations*), unfoldings explore events (*local configurations*), thus addressing (3). Explorations of repeated states and pruning of non-terminating executions is elegantly achieved in unfoldings by means of cutoff events. This solves (4) and (5).

Some of these solutions indeed seem, at present, incompatible with each other. We do not claim that the combination of POR and unfoldings immediately addresses the above problems. However, since both unfoldings and PORs share many fundamental similarities, tackling these problems in a unified framework is likely to shed light on them.

This paper lays out a DPOR algorithm on top of an unfolding structure. Our main result is a novel stateless, optimal DPOR that explores at most once every Mazurkiewicz trace, and often many fewer owing to cutoff events (cutoffs stop traces that could later branch into multiple traces). It also copes with non-terminating systems and exploits all available RAM with a *cache memory* of events, speeding up revisiting events. This provides a solution to (4), (5), (6), and a partial solution to (3). Our algorithm can alternatively be viewed as a stateless unfolding exploration, partially addressing (1) and (2).

Our result reveals DPORs as algorithms exploring an object that has richer structure than a plain directed graph. Specifically, unfoldings provide a solid notion of event *across multiple executions*, and a clear notion of conflict. Our algorithm indirectly maps important POR notions to concepts in unfolding theory.

▸ **Example 1.** We illustrate problems (3), (4), and (5), and explain how our DPOR deals with them. The following code is the skeleton of a producer-consumer program. Two concurrent producers write in, resp., `buf1` and `buf2`. The consumer accesses the buffers in sequence.

[1] Though it is a very popular one, all PORs based on persistent sets [7], for instance, are based on commutativity.

```
while (1):
  lock(m1)
  if (buf1 < MAX): buf1++
  unlock(m1)
```

```
while (1):
  lock(m2)
  if (buf2 < MAX): buf2++
  unlock(m2)
```

```
while (1):
  lock(m1)
  if (buf1 > MIN): buf1--
  unlock(m1)
  // same for m2, buf2
```

Lock and unlock operations on both mutexes `m1` and `m2` create many Mazurkiewicz traces. However, most of them have isomorphic *suffixes*, e.g., producing two items in `buf1` and consuming one reaches the same state as only producing one. After the common state, both traces explore identical behaviours and only one needs to be explored. We use cutoff events, inherited from unfolding theory [5, 3], to dynamically stop the first trace and continue only with the second. This addresses (4) and (5), and partially deals with (3). Observe that cutoff events are a form of semantic pruning, in contrast to the syntactic pruning introduced by, e.g., bounding the depth of loops, a common technique for coping with non-terminating executions in DPOR. With cutoffs, the exploration can build unreachability *proofs*, while depth bounding renders DPOR incomplete, i.e., it can only *find bugs*.

Our first step is to formulate PORs and unfoldings in the same framework. PORs are often presented for abstract execution models, while unfoldings have mostly been considered for Petri nets, where the definition is entangled with the syntax of the net. We make a second contribution here. We define, for a general execution model, event structure semantics [16] parametric on a given independence relation.

Section 2 sets up basic notions and § 3 presents our parametric event-structure semantics. In § 4 we introduce our DPOR, § 5 improves it with cutoff detection and discusses event caching. Experimental results are in § 6 and related work in § 7. We conclude in § 8. All lemmas cited along the paper and proofs of all stated results can be found in the appendixes.

2 Execution Model and Partial Order Reductions

We set up notation and recall the general ideas of POR. We consider an abstract model of (concurrent) computation. A *system* is a tuple $M := \langle \Sigma, T, \tilde{s} \rangle$ formed by a set Σ of *global states*, a set T of *transitions* and some *initial global state* $\tilde{s} \in \Sigma$. Each transition $t : \Sigma \to \Sigma$ in T is a *partial* function accounting for how the occurrence of t transforms the state of M.

A transition $t \in T$ is *enabled* at a state s if $t(s)$ is defined. Such t can *fire* at s, producing a new state $s' := t(s)$. We let $\mathit{enabl}(s)$ denote the set of transitions enabled at s. The *interleaving semantics* of M is the directed, edge-labelled graph $\mathcal{S}_M := \langle \Sigma, \to, \tilde{s} \rangle$ where Σ are the global states, $\tilde{s}$ is the initial state and $\to \subseteq \Sigma \times T \times \Sigma$ contains a triple $\langle s, t, s' \rangle$, denoted by $s \xrightarrow{t} s'$, iff t is enabled at s and $s' = t(s)$. Given two states $s, s' \in \Sigma$, and $\sigma := t_1.t_2 \dots t_n \in T^*$ (t_1 concatenated with t_2, ... until t_n), we denote by $s \xrightarrow{\sigma} s'$ the fact that there exist states $s_1, \dots, s_{n-1} \in \Sigma$ such that $s \xrightarrow{t_1} s_1, \dots, s_{n-1} \xrightarrow{t_n} s'$.

A *run* (or *interleaving*, or *execution*) of M is any sequence $\sigma \in T^*$ such that $\tilde{s} \xrightarrow{\sigma} s$ for some $s \in \Sigma$. We denote by $\mathit{state}(\sigma)$ the state s that σ reaches, and by $\mathit{runs}(M)$ the set of runs of M, also referred to as the *interleaving space*. A state $s \in \Sigma$ is *reachable* if $s = \mathit{state}(\sigma)$ for some $\sigma \in \mathit{runs}(M)$; it is a *deadlock* if $\mathit{enabl}(s) = \varnothing$, and in that case σ is called *deadlocking*. We let $\mathit{reach}(M)$ denote the set of reachable states in M. For the rest of the paper, we fix a system $M := \langle \Sigma, T, \tilde{s} \rangle$ and assume that $\mathit{reach}(M)$ is finite.

The core idea behind POR[2] is that certain transitions can be seen as commutative

[2] To be completely correct we should say "*POR that exploits the independence of transitions*".

operators, i.e., changing their order of occurrence does not change the result. Given two transitions $t, t' \in T$ and one state $s \in \Sigma$, we say that t, t' *commute at* s iff

- if $t \in \mathit{enabl}(s)$ and $s \xrightarrow{t} s'$, then $t' \in \mathit{enabl}(s)$ iff $t' \in \mathit{enabl}(s')$; and
- if $t, t' \in \mathit{enabl}(s)$, then there is a state s' such that $s \xrightarrow{t.t'} s'$ and $s \xrightarrow{t'.t} s'$.

For instance, the lock operations on `m1` and `m2` (Example 1), commute on every state, as they update different variables. Commutativity of transitions at states identifies an equivalence relation on the set $\mathit{runs}(M)$. Two runs σ and σ' of the same length are *equivalent*, written $\sigma \equiv \sigma'$, if they are the same sequence modulo swapping commutative transitions. Thus equivalent runs reach the same state. POR methods explore a fragment of $\mathcal{S}_M$ that contains at least one run in the equivalence class of each run that reaches each deadlock state. This is achieved by means of a so-called *selective search* [7]. Since employing commutativity can be expensive, PORs often use *independence relations*, i.e., sound under-approximations of the commutativity relation. In this work, partially to simplify presentation, we use unconditional independence.

Formally, an *unconditional independence relation* on M is any symmetric and irreflexive relation $\diamond \subseteq T \times T$ such that if $t \diamond t'$, then t and t' commute at *every* state $s \in \mathit{reach}(M)$. If t, t' are not independent according to $\diamond$, then they are *dependent*, denoted by $t \circledast t'$.

Unconditional independence identifies an equivalence relation $\equiv_\diamond$ on the set $\mathit{runs}(M)$. Formally, $\equiv_\diamond$ is defined as the transitive closure of the relation $\equiv^1_\diamond$, which in turn is defined as $\sigma \equiv^1_\diamond \sigma'$ iff there is $\sigma_1, \sigma_2 \in T^*$ such that $\sigma = \sigma_1.t.t'.\sigma_2$, $\sigma' = \sigma_1.t'.t.\sigma_2$, and $t \diamond t'$. From the properties of $\diamond$, one can immediately see that $\equiv_\diamond$ refines $\equiv$, i.e., if $\sigma \equiv_\diamond \sigma'$, then $\sigma \equiv \sigma'$.

Given a run $\sigma \in \mathit{runs}(M)$, the equivalence class of $\equiv_\diamond$ to which σ belongs is called the *Mazurkiewicz trace* of σ [13], denoted by $\mathcal{T}_{\diamond,\sigma}$. Each trace $\mathcal{T}_{\diamond,\sigma}$ can equivalently be seen as a labelled partial order $\mathcal{D}_{\diamond,\sigma}$, traditionally called the *dependence graph* (see [13] for a formalization), satisfying that a run belongs to the trace iff it is a linearization of $\mathcal{D}_{\diamond,\sigma}$.

Sleep sets [7] are another method for state-space reduction. Unlike selective exploration, they prune successors by looking at the past of the exploration, not the future.

3 Parametric Partial Order Semantics

An unfolding is, conceptually, a tree-like structure of partial orders. In this section, given an independence relation $\diamond$ (our parameter) and a system M, we define an unfolding semantics $\mathcal{U}_{M,\diamond}$ with the following property: each constituent partial order of $\mathcal{U}_{M,\diamond}$ will correspond to one dependence graph $\mathcal{D}_{\diamond,\sigma}$, for some $\sigma \in \mathit{runs}(M)$. For the rest of this paper, let $\diamond$ be an arbitrary unconditional independence relation on M. We use prime event structures [16], a non-sequential, event-based model of concurrency, to define the unfolding $\mathcal{U}_{M,\diamond}$ of M.

▸ **Definition 2** (LES). Given a set A, an *A-labelled event structure* (A-LES, or LES in short) is a tuple $\mathcal{E} := \langle E, <, \#, h\rangle$ where E is a set of *events*, $< \subseteq E \times E$ is a strict partial order on E, called *causality relation*, $h\colon E \to A$ labels every event with an element of A, and $\# \subseteq E \times E$ is the symmetric, irreflexive *conflict relation*, satisfying

- for all $e \in E$, $\{e' \in E \colon e' < e\}$ is finite, and (1)
- for all $e, e', e'' \in E$, if $e \# e'$ and $e' < e''$, then $e \# e''$. (2)

The *causes* of an event $e \in E$ are the set $\lceil e \rceil := \{e' \in E \colon e' < e\}$ of events that need to happen before e for e to happen. A *configuration* of $\mathcal{E}$ is any finite set $C \subseteq E$ satisfying:

- (causally closed) for all $e \in C$ we have $\lceil e \rceil \subseteq C$; (3)
- (conflict free) for all $e, e' \in C$, it holds that $\neg e \# e'$. (4)

Intuitively, configurations represent partially-ordered executions. In particular, the *local configuration* of e is the $\subseteq$-minimal configuration that contains e, i.e. $[e] := \lceil e \rceil \cup \{e\}$. We denote by $\mathit{conf}(\mathcal{E})$ the set of configurations of $\mathcal{E}$. Two events e, e' are in *immediate conflict*, $e \#^i e'$, iff $e \# e'$ and both $\lceil e \rceil \cup [e']$ and $[e] \cup \lceil e' \rceil$ are configurations. Lastly, given two LESs $\mathcal{E} := \langle E, <, \#, h \rangle$ and $\mathcal{E}' := \langle E', <', \#', h' \rangle$, we say that $\mathcal{E}$ is a *prefix* of $\mathcal{E}'$, written $\mathcal{E} \trianglelefteq \mathcal{E}'$, when $E \subseteq E'$, $<$ and $\#$ are the projections of $<'$ and $\#'$ to E, and $E \supseteq \{e' \in E' : e' < e \wedge e \in E\}$.

Our semantics will unroll the system M into a LES $\mathcal{U}_{M,\diamond}$ whose events are labelled by transitions of M. Each configuration of $\mathcal{U}_{M,\diamond}$ will correspond to the dependence graph $\mathcal{D}_{\diamond,\sigma}$ of some $\sigma \in \mathit{runs}(M)$. For a LES $\langle E, <, \#, h \rangle$, we define the *interleavings* of C as $\mathit{inter}(C) := \{h(e_1), \ldots, h(e_n) : e_i, e_j \in C \wedge e_i < e_j \implies i < j\}$. Although for arbitrary LES $\mathit{inter}(C)$ may contain sequences not in $\mathit{runs}(M)$, the definition of $\mathcal{U}_{M,\diamond}$ will ensure that $\mathit{inter}(C) \subseteq \mathit{runs}(M)$. Additionally, since all sequences in $\mathit{inter}(C)$ belong to the same trace, all of them reach the same state. Abusing the notation, we define $\mathit{state}(C) := \mathit{state}(\sigma)$ if $\sigma \in \mathit{inter}(C)$. The definition is neither well-given nor unique for arbitrary LES, but will be so for the unfolding.

We now define $\mathcal{U}_{M,\diamond}$. Each event will be inductively identified by a canonical name of the form $e := \langle t, H \rangle$, where $t \in T$ is a transition of M and H a configuration of $\mathcal{U}_{M,\diamond}$. Intuitively, e represents the occurrence of t after the *history* (or the causes) $H := \lceil e \rceil$. The definition will be inductive. The base case inserts into the unfolding a special *bottom event* $\perp$ on which every event causally depends. The inductive case iteratively extends the unfolding with one event. We define the set $\mathcal{H}_{\mathcal{E},\diamond,t}$ of candidate *histories* for a transition t in an LES $\mathcal{E}$ as the set which contains exactly all configurations H of $\mathcal{E}$ such that

- transition t is enabled at $\mathit{state}(H)$, and
- either $H = \{\perp\}$ or all $<$-maximal events e in H satisfy that $h(e) \diamondsuit t$,

where h is the labelling function in $\mathcal{E}$. Once an event e has been inserted into the unfolding, its associated transition $h(e)$ may be dependent with $h(e')$ for some e' already present and outside the history of e. Since the order of occurrence of e and e' matters, we need to prevent their occurrence within the same configuration, as configurations represent equivalent executions. So we introduce a conflict between e and e'. The set $\mathcal{K}_{\mathcal{E},\diamond,e}$ of *events conflicting* with $e := \langle t, H \rangle$ thus contains any event e' in $\mathcal{E}$ with $e' \notin [e]$ and $e \notin [e']$ and $t \diamondsuit h(e')$.

Following common practice [4], the definition of $\mathcal{U}_{M,\diamond}$ proceeds in two steps. We first define (Def. 3) the collection of all prefixes of the unfolding. Then we show that there exists only one $\trianglelefteq$-maximal element in the collection, and define it to be *the* unfolding (Def. 4).

▸ **Definition 3** (Finite unfolding prefixes)**.** The set of *finite unfolding prefixes* of M under the independence relation $\diamond$ is the smallest set of LESs that satisfies the following conditions:

1. The LES having exactly one event $\perp$, empty causality and conflict relations, and $h(\perp) := \varepsilon$ is an unfolding prefix.
2. Let $\mathcal{E}$ be an unfolding prefix containing a history $H \in \mathcal{H}_{\mathcal{E},\diamond,t}$ for some transition $t \in T$. Then, the LES $\langle E, <, \#, h \rangle$ resulting from extending $\mathcal{E}$ with a new event $e := \langle t, H \rangle$ and satisfying the following constraints is also an unfolding prefix of M:
   - for all $e' \in H$, we have $e' < e$;
   - for all $e' \in \mathcal{K}_{\mathcal{E},\diamond,e}$, we have $e \# e'$; and $h(e) := t$.

Intuitively, each unfolding prefix contains the dependence graph (configuration) of one or more executions of M (of finite length). The unfolding starts from $\perp$, the "root" of the tree, and then iteratively adds events enabled by some configuration until saturation, i.e., when no more events can be added. Observe that the number of unfolding prefixes as per Def. 3 will be finite iff all runs of M terminate. Due to lack of space, we give the definition

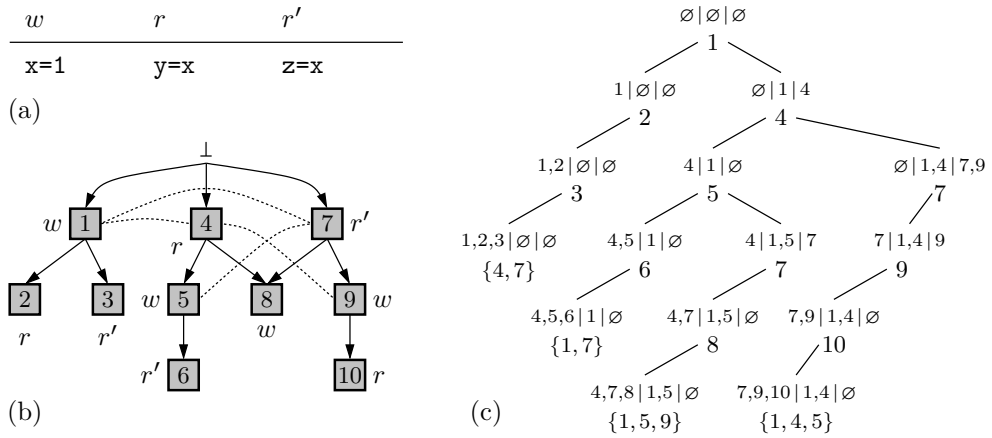

Figure 1 Running example. (a) A concurrent program; (b) its unfolding semantics. (c) The exploration performed by Alg. 1, where each node C|D|A represents one call to the function `Explore`(C, D, A). The set X underneath each leaf node is such that the value of variable U in Alg. 1 at the leaf is $U = C \cup D \cup X$. At $\varnothing|\varnothing|\varnothing$, the alternative taken is $\{4\}$, and at $4|1|\varnothing$ it is $\{7\}$.

of *infinite* unfolding prefix in App. A, as the main ideas of this section are well conveyed using only finite prefixes. In the sequel, by *unfolding prefix* we mean a finite or infinite one.

Our first task is checking that each unfolding prefix is indeed a LES (Lemma 14). Next one shows that the configurations of every unfolding prefix correspond the Mazurkiewicz traces of the system, i.e., for any configuration C, $\mathit{inter}(C) = \mathcal{T}_{\diamond,\sigma}$ for some $\sigma \in \mathit{runs}(M)$ (Lemma 16). This implies that the definition of $\mathit{inter}(C)$ and $\mathit{state}(C)$ is well-given when C belongs to an unfolding prefix. The second task is defining *the* unfolding $\mathcal{U}_{M,\diamond}$ of M. Here, we prove that the set of unfolding prefixes equipped with relation $\trianglelefteq$ forms a complete join-semilattice (Lemma 17). This implies the existence of a unique $\trianglelefteq$-maximal element:

▸ **Definition 4** (Unfolding). The *unfolding* $\mathcal{U}_{M,\diamond}$ of M under the independence relation $\diamond$ is the *unique* $\trianglelefteq$-maximal element in the set of unfolding prefixes of M under $\diamond$.

Finally we verify that the definition is well given and that the unfolding is *complete*, i.e., every run of the system is represented by a unique configuration of the unfolding.

▸ **Theorem 5.** *The unfolding $\mathcal{U}_{M,\diamond}$ exists and is unique. Furthermore, for any non-empty run σ of M, there exists a unique configuration C of $\mathcal{U}_{M,\diamond}$ such that $\sigma \in \mathit{inter}(C)$.*

▸ **Example 6** (Programs). Figure 1 (a) shows a concurrent program, where process w writes global variable and processes r and r' read it. We can associate various semantics to it. Under an empty independence relation, the unfolding would be the computation tree, where executions would be totally ordered. Considering (the unique transition of) r and r' independent, and w dependent on them, we get the unfolding shown in Fig. 1 (b).

Events are numbered from 1 to 10, and labelled with a transition. Arrows represent causality between events and dotted lines immediate conflict. The Mazurkiewicz trace of each deadlocking execution is represented by a unique $\subseteq$-maximal configuration, e.g., the run $w.r.r'$ yields configuration $\{1, 2, 3\}$, where the two possible interleavings reach the same state. The canonic name of, e.g., event 1 is $\langle w, \{\bot\}\rangle$. For event 2 it is $\langle r, \{\bot, 1\}\rangle$. Let $\mathcal{P}$ be the unfolding prefix that contains events $\{\bot, 1, 2\}$. Definition 3 can extend it with three possible events: 3, 4, and 7. Consider transition r'. Three configurations of $\mathcal{P}$ enable r': $\{\bot\}, \{\bot, 1\}$

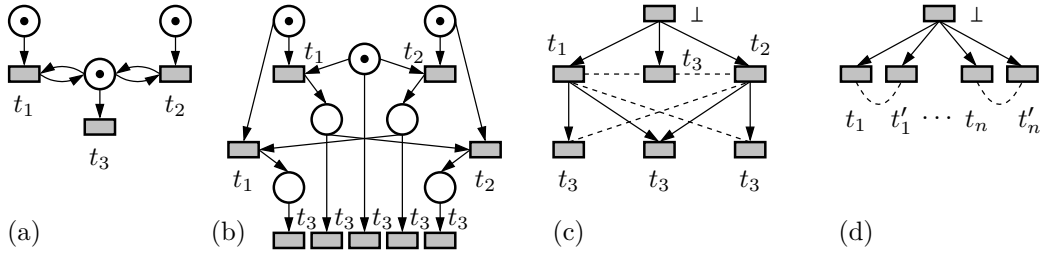

Figure 2 (a) A Petri net; (b) its classic unfolding; (c) our parametric semantics.

and $\{\bot, 1, 2\}$. But since $\neg(h(2) \otimes r')$, only the first two will be in $\mathcal{H}_{\mathcal{P},\diamond,r'}$, resulting in events $3 := \langle r', \{\bot, 1\}\rangle$ and $7 := \langle r', \{\bot\}\rangle$. Also, $\mathcal{K}_{\mathcal{P},\diamond,7}$ is $\{1\}$, as $w \otimes r'$. The 4 maximal configurations are $\{1,2,3\}$, $\{4,5,6\}$, $\{4,7,8\}$ and $\{7,9,10\}$, resp. reaching the states $\langle x, y, z\rangle = \langle 1,1,1\rangle$, $\langle 1,0,1\rangle$, $\langle 1,0,0\rangle$ and $\langle 1,1,0\rangle$, assuming that variables start at 0.

▸ **Example 7** (Comparison to Petri Net Unfoldings). In contrast to our parametric semantics, classical unfoldings of Petri nets [5] use a fixed independence relation, specifically the complement of the following one (valid only for safe nets): given two transitions t and t',

$$t \otimes_n t' \text{ iff } (t^\bullet \cap {}^\bullet t' \neq \varnothing) \text{ or } (t'^\bullet \cap {}^\bullet t \neq \varnothing) \text{ or } ({}^\bullet t' \cap {}^\bullet t \neq \varnothing),$$

where ${}^\bullet t$ and $t^\bullet$ are respectively the *preset* and *postset* of t. Classic Petri net unfoldings (of safe nets) are therefore a specific instantiation of our semantics. A well known limitation of classic unfoldings are transitions that "*read*" places, e.g., t_1 and t_2 in Fig. 2 (a). Since $t_1 \otimes_n t_2$, the classic unfolding, Fig. 2 (b), sequentializes all their occurrences. A solution to this is the so-called *place replication (PR) unfolding* [15], or alternatively *contextual unfoldings* (which anyway internally are of asymptotically the same size as the PR-unfolding).

This problem vanishes with our parametric unfolding. It suffices to use a dependency relation $\otimes'_n \subset \otimes_n$ that makes transitions that "*read*" common places independent. The result is that our unfolding, Fig. 2 (c), can be of the same size as the PR-unfolding, i.e., exponentially more compact than the classic unfolding. For instance, when Fig. 2 (a) is generalized to n *reading* transitions, the classic unfolding would have $\mathcal{O}(n!)$ copies of t_3, while ours would have $\mathcal{O}(2^n)$. The point here is that our semantics naturally accommodate a more suitable notion of independence without resorting to specific ad-hoc tricks.

Furthermore, although this work is restricted to *unconditional* independence, we conjecture that an adequately restricted *conditional* dependence would suffice, e.g., the one of [12]. Gains achieved in such setting would be difficult with classic unfoldings.

4 Stateless Unfolding Exploration Algorithm

We present a DPOR algorithm to explore an arbitrary event structure (e.g., the one of § 3) instead of sequential executions. Our algorithm explores one configuration at a time and organizes the exploration into a binary tree. Figure 1 (c) shows an example. The algorithm is optimal [2], in the sense that no configuration is ever visited twice in the tree.

For the rest of the paper, let $\mathcal{U}_{\diamond,M} := \langle E, <, \#, h\rangle$ be the unfolding of M under $\diamond$, which we abbreviate as $\mathcal{U}$. For this section we assume that $\mathcal{U}$ is finite, i.e., that all computations of M terminate. This is only to ease presentation, we relax this assumption in § 5.2.

Algorithm 1: An unfolding-based POR exploration algorithm.

```
1  Initially, set U := {⊥}, set G := ∅, and call Explore({⊥}, ∅, ∅).
2  Procedure Explore(C, D, A)
3      Extend(C)
4      if en(C) = ∅ return
5      if A = ∅
6          Choose e from en(C)
7      else
8          Choose e from A ∩ en(C)
9      Explore(C ∪ {e}, D, A ∖ {e})
10     if ∃J ∈ Alt(C, D ∪ {e})
11         Explore(C, D ∪ {e}, J ∖ C)
12     Remove(e, C, D)
13 Procedure Extend(C)
14     Add ex(C) to U
15 Procedure Remove(e, C, D)
16     Move {e} ∖ Q_{C,D,U} from U to G
17     foreach ê ∈ #^i_U(e)
18         Move [ê] ∖ Q_{C,D,U} from U to G
```

We give some new definitions. Let C be a configuration of $\mathcal{U}$. The *extensions* of C, written $ex(C)$, are all those events outside C whose causes are included in C. Formally, $ex(C) := \{e \in E : e \notin C \wedge \lceil e \rceil \subseteq C\}$. We let $en(C)$ denote the set of events *enabled* by C, i.e., those corresponding to the transitions enabled at $state(C)$, formally defined as $en(C) := \{e \in ex(C) : C \cup \{e\} \in conf(\mathcal{U})\}$. All those events in $ex(C)$ which are not in $en(C)$ are the *conflicting extensions*, $cex(C) := \{e \in ex(C) : \exists e' \in C,\ e \#^i e'\}$. Clearly, sets $en(C)$ and $cex(C)$ partition the set $ex(C)$. Lastly, we define $\#^i(e) := \{e' \in E : e \#^i e'\}$, and $\#^i_U(e) := \#^i(e) \cap U$. The difference between both is that $\#^i(e)$ contains events from *anywhere* in the unfolding structure, while $\#^i_U(e)$ can only *see* events in U.

The algorithm is given in Alg. 1. `Explore`(C, D, A), the main procedure, is given the configuration that is to be explored as the parameter C. The parameter D (for *disabled*) is the set of set of events that have already been explored and prevents that `Explore()` repeats work. It can be seen as a *sleep set* [7]. Set A (for *add*) is occasionally used to guide the direction of the exploration.

Additionally, a global set U stores all events presently known to the algorithm. Whenever some event can safely be discarded from memory, `Remove` will move it from U to G (for *garbage*). Once in G, it can be discarded at any time, or be preserved in G in order to save work when it is re-inserted in U. Set G is thus our *cache memory* of events.

The key intuition in Alg. 1 is as follows. A call to `Explore`(C, D, A) visits all maximal configurations of $\mathcal{U}$ which contain C and do not contain D; and the first one explored will contain $C \cup A$. Figure 1 (c) gives one execution, tree nodes are of the form C|D|A.

The algorithm first updates U with all extensions of C (procedure `Extend`). If C is a maximal configuration, then there is nothing to do, it backtracks. If not, it chooses an event in U enabled at C, using the function `en`$(C) := en(C) \cap U$. If A is empty, any enabled event can be taken. If not, A needs to be explored and e must come from the intersection. Next it makes a recursive call (left subtree), where it explores *all* configurations containing all events in $C \cup \{e\}$ and no event from D. Since `Explore`(C, D, A) had to visit all maximal configurations containing C, it remains to visit those containing C but not e, but only if there exists at least one! Thus, we determine whether $\mathcal{U}$ has a maximal configuration that contains C, does not contain D and does not contain e. Function `Alt` will return a set of events that witness the existence of such configuration (iff one exists). If one exists, we make

a second recursive call (right subtree). Formally, we call such witness an *alternative*:

▸ **Definition 8** (Alternatives). Given a set of events $U \subseteq E$, a configuration $C \subseteq U$, and a set of events $D \subseteq U$, an *alternative* to D after C is any configuration $J \subseteq U$ satisfying that

- $C \cup J$ is a configuration (5)
- for all events $e \in D$, there is some $e' \in C \cup J$ such that $e' \in \#^i_U(e)$. (6)

Function `Alt(`X, Y`)` returns all alternatives (in U) to Y after X. Notice that it is called as `Alt(`$C, D \cup \{e\}$`)` from Alg. 1. Any returned alternative J witnesses the existence of a maximal configuration C' (constructed by arbitrarily extending $C \cup J$) where $C' \cap (D \cup \{e\}) = \varnothing$.

Although `Alt` reasons about maximal configurations of $\mathcal{U}$, thus potentially about events which have not yet been seen, it can only look at events in U. So the set U needs to be large enough to contain enough *conflicting events* to satisfy (6). Perhaps surprisingly, it suffices to store only events seen (during the past exploration) in immediate conflict with C and D. Consequently, when the algorithm calls `Remove`, to clean from U events that are no longer necessary (i.e., necessary to find alternatives in the future), it needs to preserve at least those conflicting events. Specifically, `Remove` will preserve in U the following events:

$$Q_{C,D,U} := C \cup D \cup \bigcup_{e \in C \cup D, e' \in \#^i_U(e)} [e'].$$

That is, events in C, in D and events in conflict with those. An alternative definition that makes $Q_{C,D,U}$ smaller would mean that `Remove` discards more events, which could prevent a future call to `Alt` from discovering a maximal configuration that needs to be explored.

We focus now on the correctness of Alg. 1. Every call to `Explore(`C, D, A`)` explores a tree, where the recursive calls at lines line 9 and line 11 respectively explore the left and right subtrees (proof in Corollary 25). Tree nodes are tuples $\langle C, D, A\rangle$ corresponding to the arguments of calls to `Explore`, cf. Fig. 1. We refer to this object as the *call tree*. For every node, both C and $C \cup A$ are configurations, and $D \subseteq ex(C)$ (Lemma 18). As the algorithm goes down in the tree it monotonically increases the size of either C or D. Since $\mathcal{U}$ is finite, this implies that the algorithm terminates:

▸ **Theorem 9** (Termination). *Regardless of its input, Alg. 1 always stops.*

Next we check that Alg. 1 never visits twice the same configuration, which is why it is called an *optimal* POR [2]. We show that for every node in the call tree, the set of configurations in the left and right subtrees are disjoint (Lemma 24). This implies that:

▸ **Theorem 10** (Optimality). *Let $\tilde{C}$ be a maximal configuration of $\mathcal{U}$. Then* `Explore(`$\cdot,\cdot,\cdot$`)` *is called* at most once *with its first parameter being equal to $\tilde{C}$.*

Parameter A of `Explore` plays a central role in making Alg. 1 optimal. It is necessary to ensure that, once the algorithm decides to explore some alternative J, such an alternative is visited first. Not doing so makes it possible to extend C in such a way that no maximal configuration can ever avoid including events in D. Such a configuration, referred as a *sleep-set blocked* execution in [2], has already been explored before.

Finally, we ensure that Alg. 1 visits every maximal configuration of $\mathcal{U}$. This essentially reduces to showing that it makes the second recursive call, line 11, whenever there exists some unexplored maximal configuration not containing $D \cup \{e\}$. The difficulty of proving so (Lemma 27) comes from the fact that Alg. 1 *only* sees events in U. Due to space constraints, we omit an additional result on the memory consumption, cf. App. B.5.

▸ **Theorem 11** (Completeness). *Let $\tilde{C}$ be a maximal configuration of $\mathcal{U}$. Then* `Explore(`$\cdot,\cdot,\cdot$`)` *is called* at least once *with its first parameter being equal to $\tilde{C}$.*

5 Improvements

5.1 State Caching

Stateless model checking algorithms explore only one configuration of $\mathcal{U}$ at a time, thus potentially under-using remaining available memory. A desirable property for an algorithm is the capacity to exploit all available memory without imposing the liability of actually requiring it. The algorithm in § 4 satisfies this property. The set G, storing events discarded from U, can be cleaned at discretion, e.g., when the memory is approaching full utilisation. Events cached in G are exploited in two different ways.

First, whenever an event in G shall be included again in U, we do not need to reconstruct it in memory (causality, conflicts, etc.). In extreme cases, this might happen frequently. Second, using the result of the next section, cached events help prune the number of maximal configurations to visit. This means that our POR potentially visits *fewer* final states than the number of configurations of $\mathcal{U}$, thus conforming to the requirements of a *super-optimal DPOR*. The larger G is, the fewer configurations will be explored.

5.2 Non-Acyclic State Spaces

In this section we remove the assumption that $\mathcal{U}_{M,\diamond}$ is finite. We employ the notion of cutoff events [14]. While cutoffs are a standard tool for unfolding pruning, their application to our framework brings unexpected problems.

The core question here is preventing Alg. 1 from getting stuck in the exploration of an infinite configuration. We need to create the illusion that maximal configurations are finite. We achieve this by substituting procedure `Extend` in Alg. 1 with another procedure `Extend'` that operates as `Extend` except that it only adds to U an event from $e \in ex(C)$ if the predicate $\mathsf{cutoff}(e, U, G)$ evaluates to false. We define $\mathsf{cutoff}(e, U, G)$ to hold iff there exists some event $e' \in U \cup G$ such that

$$state([e]) = state([e']) \quad \text{and} \quad |[e']| < |[e]|. \tag{7}$$

We refer to e' as the *corresponding* event of e, when it exists. This definition declares e cutoff as function of U and G. This has important consequences. An event e could be declared cutoff while exploring one maximal configuration and non-cutoff while exploring the next, as the corresponding event might have disappeared from $U \cup G$. This is in stark contrast to the classic unfolding construction, where events are declared cutoffs *once and for all*. The main implication is that the standard argument [14, 5, 3] invented by McMillan for proving completeness fails. We resort to a completely different argument for proving completeness of our algorithm (see App. C.1), which we are forced to skip in view of the lack of space.

We focus now on the correction of Alg. 1 using `Extend'` instead of `Extend`. A *causal cutoff* is any event e for which there is some $e' \in [e]$ satisfying (7). It is well known that causal cutoffs define a finite prefix of $\mathcal{U}$ as per the classic saturation definition [3]. Also, $\mathsf{cutoff}(e, U, G)$ always holds for causal cutoffs, regardless of the contents of U and G. This means that the modified algorithm can only explore configurations from a finite prefix. It thus necessarily terminates. As for optimality, it is unaffected by the use of cutoffs, existing proofs for Alg. 1 still work. Finally, for completeness we prove the following result, stating that local reachability (e.g., fireability of transitions of M) is preserved:

▸ **Theorem 12** (Completeness). *For any reachable state $s \in reach(M)$, Alg. 1 updated with the cutoff mechanism described above explores one configuration C such that for some $C' \subseteq C$ it holds that $state(C') = s$.*

Table 1 Programs with acyclic state space. Columns are: $|P|$: nr. of threads; $|I|$: nr. of explored traces; $|B|$: nr. of sleep-set blocked executions; $t(s)$: running time; $|E|$: nr. of events in $\mathcal{U}$; $|E_{\text{cut}}|$: nr. of cutoff events; $|\Omega|$: nr. of maximal configurations; $\langle|U_\Omega|\rangle$: avg. nr. of events in U when exploring a maximal configuration. A $*$ marks programs containing bugs. <7K reads as "*fewer than 7000*".

Benchmark		Nidhugg			Poet (without cutoffs)				Poet (with cutoffs)				
Name	$\|P\|$	$\|I\|$	$\|B\|$	$t(s)$	$\|E\|$	$\|\Omega\|$	$\langle\|U_\Omega\|\rangle$	$t(s)$	$\|E\|$	$\|E_{\text{cut}}\|$	$\|\Omega\|$	$\langle\|U_\Omega\|\rangle$	$t(s)$
Stf	3	6	0	0.06	121	6	79	0.04	121	0	6	79	0.06
Stf$*$	3	–	–	0.05	–	–	–	0.02	–	–	–	–	0.03
Spin08	3	84	0	0.08	2974	84	1506	2.04	2974	0	84	1506	2.93
Fib	3	8953	0	3.36	<185K	8953	92878	305	<185K	0	8953	92878	704
Fib$*$	3	–	–	0.74	–	–	–	81.0	–	–	–	–	133
Ccnf(9)	9	16	0	0.05	49	16	46	0.07	49	0	16	46	0.06
Ccnf(17)	17	256	0	0.15	97	256	94	5.76	97	0	256	94	6.09
Ccnf(19)	19	512	0	0.28	109	512	106	22.5	109	0	512	106	22.0
Ssb	5	4	2	0.05	48	4	38	0.03	46	1	4	37	0.03
Ssb(1)	5	22	14	0.06	245	23	143	0.11	237	4	23	140	0.11
Ssb(3)	5	169	67	0.12	2798	172	1410	3.51	1179	48	90	618	0.90
Ssb(4)	5	336	103	0.15	<7K	340	3333	20.3	2179	74	142	1139	2.07
Ssb(8)	5	2014	327	0.85	<67K	2022	32782	4118	<12K	240	470	6267	32.1

Lastly, we note that this cutoff approach imposes no liability on what events shall be kept in the prefix, set G can be cleaned at discretion. Also, redefining (7) to use adequate orders [5] is straightforward, cf. App. C.1 (in our proofs we actually assume adequate orders).

6 Experiments

As a proof of concept, we implemented our algorithm in a new explicit-state model checker baptized Poet (Partial Order Exploration Tool).[3] Written in Haskell, a lazy functional language, it analyzes programs from a restricted fragment of the C language and supports POSIX threads. The analyzer accepts deterministic programs, implements a variant of Alg. 1 where the computation of the alternatives is memoized, and supports cutoffs events with the criteria defined in § 5.

We ran Poet on a number of multi-threaded C programs. Most of them are adapted from benchmarks of the Software Verification Competition [17]; others are used in related works [8, 19, 2]. We investigate the characteristics of average program unfoldings (depth, width, etc.) as well as the frequency and impact of cutoffs on the exploration. We also compare Poet with Nidhugg [1], a state-of-the-art stateless model checking for multi-threaded C programs that implements Source-DPOR [2], an efficient but non-optimal DPOR. All experiments were run on an Intel Xeon CPU with 2.4 GHz and 4 GB memory. Tables 1 and 2 give our experimental data for programs with acyclic and non-acyclic state spaces, respectively.

For programs with acyclic state spaces (Table 1), Poet with and without cutoffs seems to perform the same exploration when the unfolding has no cutoffs, as expected. Furthermore, the number of explored executions also coincides with Nidhugg when the latter reports 0 sleep-set blocked executions (cf., § 4), providing experimental evidence of Poet's optimality.

The unfoldings of most programs in Table 1 do not contain cutoffs. All these programs are deterministic, and many of them highly sequential (Stf, Spin08, Fib), features known to make cutoffs unlikely. Ccnf(n) are concurrent programs composed of $n-1$ threads where thread i and $i+1$ race on writing one variable, and are independent of all remaining

[3] Source code and benchmarks available from: http://www.cs.ox.ac.uk/people/marcelo.sousa/poet/.

Table 2 Programs with non-terminating executions. Column b is the loop bound. The value is chosen based on experiments described in [1].

Benchmark		NIDHUGG				POET (with cutoffs)				
Name	$\|P\|$	b	$\|I\|$	$\|B\|$	$t(s)$	$\|E\|$	$\|E_{\text{cut}}\|$	$\|\Omega\|$	$\langle\|U_\Omega\|\rangle$	$t(s)$
SZYMANSKI	3	-	103	0	0.07	1121	313	159	591	0.36
DEKKER	3	10	199	0	0.11	217	14	21	116	0.07
LAMPORT	3	10	32	0	0.06	375	28	30	208	0.12
PETERSON	3	10	266	0	0.11	175	15	20	100	0.05
PGSQL	3	10	20	0	0.06	51	8	4	40	0.03
RWLOCK	5	10	2174	14	0.83	<7317	531	770	3727	12.29
RWLOCK(2)*	5	2	-	-	7.88	-	-	-	-	0.40
PRODCONS	4	5	756756	0	332.62	3111	568	386	1622	5.00
PRODCONS(2)	4	5	63504	0	38.49	640	25	15	374	1.61

threads. Their unfoldings resemble Fig. 2 (d), with $2^{(n-1)/2}$ traces but only $\mathcal{O}(n)$ events. Saturation-based unfolding methods would win here over both NIDHUGG and POET.

In the SSB benchmarks, NIDHUGG encounters sleep-set blocked executions, thus performing sub-optimal exploration. By contrast, POET finds many cutoff events and achieves a *super-optimal* exploration, exploring fewer traces than both POET without cutoffs and NIDHUGG. The data shows that this *super-optimality* results in substantial savings in runtime.

For non-acyclic state spaces (Table 2), unfoldings are infinite. We thus compare POET with cutoffs and NIDHUGG with a loop bound. Hence, while NIDHUGG performs bounded model checking, POET does complete verification. The benchmarks include classical mutual exclusion protocols (SZYMANSKI, SEKKER, LAMPORT and PETERSON), where NIDHUGG is able to leverage an important static optimization that replaces each spin loop by a load and assume statement [1]. Hence, the number of traces and maximal configurations is not comparable. Yet POET, which could also profit from this static optimization, achieves a significantly better reduction thanks to cutoffs alone. Cutoffs dynamically prune redundant unfolding branches and arguably constitute a more robust approach than the load and assume syntactic substitution. The substantial reduction in number of explored traces, several orders of magnitude in some cases, translates in clear runtime improvements. Finally, in our experiments, both tools were able to successfully discover assertion violations in STF*, FIB* and RWLOCK(2)*.

In our experiments, POET's average maximal memory consumption (measured in events) is roughly half of the size of the unfolding. We also notice that most of these unfoldings are quite narrow and deep ($|E_{\text{cut}}| \div |E|$ is low) when compared with standard benchmarks for Petri nets. This suggests that they could be amenable for saturation-based unfolding verification, possibly pointing the opportunity of applying these methods in software verification.

7 Related Work

This work focuses on explicit-state POR, as opposed to symbolic POR techniques exploited inside SAT solvers, e.g., [11, 8]. Early POR statically computed the necessary transitions to fire at every state [18, 7]. Flanagan and Godefroid [6] first proposed to compute persistent sets dynamically (DPOR). However, even when combined with sleep sets [7], DPOR was still unable to explore exactly one interleaving per Mazurkiewicz trace. Abdulla et al. [2, 1] recently proposed the first solution to this, using a data structure called wakeup trees. Their DPOR is thus optimal (ODPOR) in this sense.

Unlike us, ODPOR operates on an interleaved execution model. Wakeup trees store

chains of dependencies that assist the algorithm in reversing races throughly. Technically, each branch roughly correspond to one of our alternatives. According to [2], constructing and managing wakeup trees is expensive. This seems to be related with the fact that wakeup trees store canonical linearizations of configurations, and need to canonize executions before inserting them into the tree to avoid duplicates. Such checks become simple linear-time verifications when seen as partial-orders. Our alternatives are computed dynamically and exploit these partial orders, although we do not have enough experimental data to compare with wakeup trees. Finally, our algorithm is able to visit up to exponentially fewer Mazurkiewicz traces (owing to cutoff events), copes with non-terminating executions, and profits from state-caching. The work in [2] has none of these features.

Combining DPOR with stateful search is challenging [20]. Given a state s, DPOR relies on a complete exploration from s to determine the necessary transitions to fire from s, but such exploration could be pruned if a state is revisited, leading to unsoundness. Combining both methods requires addressing this difficulty, and two works did it [20, 19], but for non-optimal DPOR. By contrast, incorporating cutoff events into Alg. 1 was straightforward.

Classic, saturation-based unfolding algorithms are also related [14, 5, 3, 10]. They are inherently stateful, cannot discard events from memory, but explore events instead of configurations, thus may do exponentially less work. They can furthermore guarantee that the number of explored events will be at most the number of reachable states, which at present seems a difficult goal for PORs. On the other hand, finding the events to extend the unfolding is computationally harder. In [10], Kähkönen and Heljanko use unfoldings for concolic testing of concurrent programs. Unlike ours, their unfolding is not a semantics of the program, but rather a means for discovering all concurrent program paths.

While one goal of this paper is establishing an (optimal) POR exploiting the same commutativity as some non-sequential semantics, a longer-term goal is building formal connections between the latter and PORs. Hansen and Wang [9] presented a characterization of (a class of) stubborn sets [18] in terms of configuration structures, another non-sequential semantics more general than event structures. We shall clarify that while we restrict ourselves to commutativity-based PORs, they attempt a characterization of stubborn sets, which do not necessarily rely on commutativity.

8 Conclusions

In the context of commutativity-exploiting POR, we introduced an optimal DPOR that leverages on cutoff events to prune the number of explored Mazurkiewicz traces, copes with non-terminating executions, and uses state caching to speed up revisiting events. The algorithm provides a new view to DPORs as algorithms exploring an object with richer structure. In future work, we plan exploit this richer structure to further reduce the number of explored traces for both PORs and saturation-based unfoldings.

References

1 Parosh Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless Model Checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, number 9035 in LNCS, pages 353–367. Springer, 2015.

2 Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *Principles of Programming Languages (POPL)*, pages 373–384. ACM, 2014.

3 Blai Bonet, Patrik Haslum, Victor Khomenko, Sylvie Thiébaux, and Walter Vogler. Recent advances in unfolding technique. *Theoretical Comp. Science*, 551:84–101, September 2014.

4 Javier Esparza and Keijo Heljanko. *Unfoldings – A Partial-Order Approach to Model Checking*. EATCS Monographs in Theoretical Computer Science. Springer, 2008.

5 Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20:285–310, 2002.

6 Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Principles of Programming Languages (POPL)*, pages 110–121. ACM, 2005.

7 Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.

8 Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian partial-order reduction. In *Model Checking Software (SPIN)*, volume 4595 of *LNCS*, pages 95–112. Springer, 2007.

9 Henri Hansen and Xu Wang. On the origin of events: branching cells as stubborn sets. In *Proc. International Conference on Application and Theory of Petri Nets and Concurrency (ICATPN)*, volume 6709 of *LNCS*, pages 248–267. Springer, 2011.

10 Kari Kähkönen and Keijo Heljanko. Testing multithreaded programs with contextual unfoldings and dynamic symbolic execution. In *Application of Concurrency to System Design (ACSD)*, pages 142–151. IEEE, 2014.

11 Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *Computer Aided Verification (CAV)*, volume 5643 of *LNCS*, pages 398–413. Springer, 2009.

12 Shmuel Katz and Doron Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101(2):337–359, 1992.

13 Antoni Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 278–324. Springer, 1987.

14 K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of async. circuits. In *Proc. CAV'92*, volume 663 of *LNCS*, pages 164–177. Springer, 1993.

15 Ugo Montanari and F. Rossi. Contextual nets. *Acta Informatica*, 32(6):545–596, 1995.

16 Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13(1):85–108, 1981.

17 http://sv-comp.sosy-lab.org/2015/.

18 Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, number 483 in LNCS, pages 491–515. Springer, 1991.

19 Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient stateful dynamic partial order reduction. In *Model Checking Software (SPIN)*, volume 5156 of *LNCS*, pages 288–305. Springer, 2008.

20 Xiaodong Yi, Ji Wang, and Xuejun Yang. Stateful dynamic partial-order reduction. In *Formal Methods and Sw. Eng.*, number 4260 in LNCS, pages 149–167. Springer, 2006.

A Proofs: Unfolding Semantics

In § 3 we defined the set of finite unfolding prefixes of M under the independence relation $\diamond$. If M has only terminating executions, i.e., all elements in $runs(M)$ are finite, then all unfolding prefixes are finite. However, if it has non-terminating executions, then we need to also consider its infinite unfolding prefixes. We will achieve this in Def. 13. First we need a technical definition and some results about it. Let

$$F := \{\langle E_1, <_1, \#_1, h_1\rangle, \langle E_2, <_2, \#_2, h_2\rangle, \ldots\}$$

be a finite or infinite set of unfolding prefixes of M under $\diamond$. We define the *union* of all of them as the LES $union(F) := \langle E, <, \#, h\rangle$, where

$$E := \bigcup_{1\leq i} E_i \qquad < := \bigcup_{1\leq i} <_i \qquad h := \bigcup_{1\leq i} h_i,$$

and $\#$ is the $\subseteq$-minimal relation on $E \times E$ that satisfies (2) and such that $e \# e'$ holds for any two events $e, e' \in E$ if

$$e \notin [e'] \text{ and } e' \notin [e] \text{ and } h(e) \otimes h(e'). \tag{8}$$

Since every element of F is a LES, clearly $union(F)$ is also a LES, (1) and (2) are trivially satisfied. Notice that all events in $E_1, E_2, E_3, \ldots$ are pairs of the form $\langle t, H\rangle$, and the union of two or more E_i's will merge many *equal events*. Indeed, two events $e_1 := \langle t_1, H_1\rangle$ and $e_2 := \langle t_2, H_2\rangle$ are equal iff $t_1 = t_2$ and $H_1 = H_2$.

▸ **Definition 13** (Unfolding prefixes, finite or infinite)**.** The set of *unfolding prefixes* of M under the independence relation $\diamond$ contains all finite unfolding prefixes, as defined by Def. 3, together with those constructed by:

- For any infinite set X of unfolding prefixes, $union(X)$ is also an unfolding prefix.

Our first task is verifying that each unfolding prefix is indeed a LES. Conditions (1) and (2) are satisfied by construction. We verify the following:

▸ **Lemma 14.** *For any unfolding prefix $\mathcal{P} := \langle E, <, \#, h\rangle$ we have the following:*

1. *The relation $<$ is a strict partial order.*
2. *The relation $\#$ is irreflexive.*

Proof. Assume that $\mathcal{P}$ is finite. This means that it has been constructed with Def. 3. We prove both statements by induction.

Base case. The prefix containing only $\perp$ trivially satisfies both statements.

Step case. We prove both statements separately. Clearly $e < e$ does not hold, as every event introduced by Def. 3 is a causal successor of only events that were already present in the unfolding prefix. Furthermore, the insertion of an event does not change the causal relations existing in the preceding unfolding prefix. The relation $<$ is also transitive, as the history of a configuration is causally closed.

As for the second statement, we prove it by contradiction. Assume that $e \# e$ and that e has been inserted into $\mathcal{P}$ by applying Def. 3 to the prefix $\mathcal{P}'$. Clearly, $e \notin \mathcal{D}_{\mathcal{P}',e}$, so the conflict has not been inserted when extending $\mathcal{P}$ with e. It must be the case, then, that Def. 3 has inserted another event e' in E after inserting e, and that $e' \in \lceil e\rceil$ and $e' \# e$. This is also not possible since, by definition, when inserting e' on a prefix $\mathcal{P}''$ no causal successor of e' can be present in $\mathcal{D}_{\mathcal{P}'',e'}$.

Assume now that $\mathcal{P}$ is not finite. Then it is the union of an infinite family of finite unfolding prefixes, each one of them satisfy the above. We prove again both statements separately.

First statement. For any event $e := \langle t, H\rangle \in E$, necessarily $e < e$ cannot hold, as e comes from some of the finite prefixes. Now, if e belongs to several finite prefixes, by construction they agree on which events are causal predecessors of e. If the union contains a cycle

$$e_1 < e_2 < \ldots < e_n < e_1,$$

then all n events are present in any finite prefix to which e_n belongs. As a result all of them are in $\lceil e_n \rceil$, which is clearly impossible.

Second statement. It cannot be the case that $e \# e$ in $\mathcal{P}$ but $\neg(e \# e)$ in any finite prefix that gives rise to $\mathcal{P}$, by definition of $\mathit{union}(\cdot)$. So since $\neg(e \# e)$ holds for any finite prefix, then $\neg(e \# e)$ holds for $\mathcal{P}$. ◂

We now need to prove some facts about $\mathit{union}(\cdot)$.

▸ **Lemma 15.** *If F is a finite set of unfolding prefixes constructed by Def. 3, then $\mathit{union}(F)$ is also a finite prefix constructed by Def. 3.*

Proof. (Sketch). The proof proceeds by induction on the size n of F. If $n = 1$ then it is easy to see that the union is a finite prefix (observe that $\mathit{union}(\cdot)$ "discards" the original conflict relation and substitutes it for a new one).

The inductive step reduces to showing that the union of two prefixes is a prefix, as

$$\mathit{union}(F' \cup \{\mathcal{P}\}) = \mathit{union}(\{\mathit{union}(F')\} \cup \{\mathcal{P}\}).$$

To show this, let $\mathcal{P}_1 := \langle E_1, <_1, \#_1, h_1\rangle$ and $\mathcal{P}_2 := \langle E_2, <_2, \#_2, h_2\rangle$ be two unfolding prefixes. To show that $\mathit{union}(\{\mathcal{P}_1, \mathcal{P}_2\})$ is an unfolding prefix we proceed again by induction in the size m of $E_2 \setminus E_1$. If $m = 0$ then $\mathcal{P}_2 \trianglelefteq \mathcal{P}_1$ and we are done. If not one can select a $<$-maximal event $e := \langle t, H\rangle$ from $E_2 \setminus E_1$, remove it from $\mathcal{P}_2$, and the resulting prefix $\mathcal{P}_2'$ is such that $\mathcal{P}_3 := \mathit{union}(\mathcal{P}_1, \mathcal{P}_2')$ is a finite prefix generated by Def. 3. Now Def. 3 can extend $\mathcal{P}_3$ with e, as H is by hypothesis a configuration of $\mathcal{P}_3$ that enables t and so on. Finally, one shows that the causality, label, and conflict relation that Def. 3 and the definition of $\mathit{union}(\cdot)$ will attach to e coincide. ◂

Next we show that every configuration of every unfolding prefix corresponds to some Mazurkiewicz trace of the system:

▸ **Lemma 16.** *Let $\mathcal{P}$ be an unfolding prefix of M under $\diamond$. Given any configuration C of $\mathcal{P}$, it holds that $\mathit{inter}(C) \subseteq \mathit{runs}(M)$. Furthermore, for any two runs $\sigma_1, \sigma_2 \in \mathit{inter}(C)$, we have $\mathit{state}(\sigma_1) = \mathit{state}(\sigma_2)$.*

Proof. Let $\mathcal{P} := \langle E, <, \#, h\rangle$ be the prefix, with $h : E \to T$. Let C be a configuration of $\mathcal{P}$. In this proof we will assume that $\mathcal{P}$ is finite. This is because, of the following two facts:

- Assume that $\mathcal{P} = \mathit{union}(F)$, where $F := \{\mathcal{P}_1, \mathcal{P}_2, \ldots\}$ is an infinite collection of finite prefixes. Only finitely many prefixes in F contain events of C, as C is finite.
- By Lemma 15, the $\mathit{union}(\cdot)$ of finitely many prefixes is a finite prefix generated by Def. 3.

So if $\mathcal{P}$ is infinite, by the above, we can find a finite prefix $\mathcal{P}'$, generated by Def. 3, and which contains C. Since the arguments we make in the sequel only concern events in C, proving the lemma in $\mathcal{P}'$ is equivalent to proving it in $\mathcal{P}$.

So w.l.o.g. we assume that $\mathcal{P}$ is a finite unfolding. The proof is by structural induction on the set of unfolding prefixes ordered by the prefix relation $\trianglelefteq$.

Base case. Assume that $\mathcal{P}$ has been produced by the first rule of Def. 3. Then $E = \{\bot\}$ and the lemma trivially holds.

Inductive step. Assume $\mathcal{P}$ that has been produced by the application of the second rule of Def. 3 to the unfolding prefix $\mathcal{P}'$, and let e be the only event in $\mathcal{P}$ but not in $\mathcal{P}'$. Also, assume that the lemma holds for $\mathcal{P}'$.

Only two things are possible: $e \in C$ or $e \notin C$. In the second case, C is a configuration of $\mathcal{P}'$ and we are done, so assume that $e \in C$. Necessarily e is a $<$-maximal event in C. Let $\sigma \in \mathit{inter}(C)$ be an interleaving of C, and let $C := \{e_1, \ldots, e_n\}$. W.l.o.g., assume that σ is of the form

$$\sigma = h(e_1), \ldots, h(e_n)$$

and that $e_i = e$. Clearly, the causes $\lceil e \rceil$ of e are a subset of the events $\{e_1, \ldots, e_{i-1}\}$. Since, by definition of $\mathit{inter}(\cdot)$, $\{e_1, \ldots, e_{i-1}\}$ is a configuration and it does not include e, it is necessarily a configuration of $\mathcal{P}'$. Thus, by applying the induction hypothesis we know that the sequence

$$h(e_1), \ldots, h(e_{i-1})$$

is an execution of M and produces the same global state as another execution that first fires all events in $\lceil e \rceil$ and then all remaining events in $\{e_1, \ldots, e_{i-1}\}$. This means that σ is an execution of M iff the sequence

$$\sigma' := \sigma''.h(f_1) \ldots h(f_k).h(e).h(g_1) \ldots h(g_l)$$

is an execution of M, where $\sigma'' \in \mathit{inter}(\lceil e \rceil)$, $\{f_1, \ldots, f_k\} = \{e_1, \ldots, e_{i-1}\} \setminus \lceil e \rceil$, and $g_1 = e_{i+1}$, $\ldots$, $g_l = e_n$.

Now we will show that the sequence $\sigma''.h(f_1) \ldots h(f_k).h(e)$, which is a prefix of σ', is an execution. From Def. 3 we know that σ'' enables $h(e)$, and from the induction hypothesis we also know that σ'' enables $h(f_1)$. Since $\neg f_1 \# e$ and $f_1 \notin \lceil e \rceil$, from Def. 3 we know that $h(f_1) \diamond h(e)$, i.e., the transitions associated to both events commute (at all states). Since both $h(f_1)$ and $h(e)$ are enabled at $\mathit{state}(\sigma'')$, then $\sigma''.h(f_1).h(e)$ is a run. Again, the run $\sigma''.h(f_1)$ enables both $h(e)$ and $h(f_2)$, and for similar reasons $h(e) \diamond h(f_2)$, so we know that $\sigma''.h(f_1).h(f_2).h(e)$ is a run. Iterating this argument k times one can prove that

$$\tilde{\sigma} := \sigma''.h(f_1) \ldots h(f_k).h(e)$$

is indeed an execution.

The next step is proving that the execution $\tilde{\sigma}$ can be continued by firing the sequence of transitions $h(g_1), \ldots, h(g_l)$. The argument here is quite similar as before, but slightly different. It is easy to see that $h(e) \diamond h(g_j)$ for $j \in \{1, \ldots, l\}$. Since $\tilde{\sigma}$ enables both $h(e)$ and $h(g_1)$, and both commute at $\mathit{state}(\tilde{\sigma})$, then necessarily $\tilde{\sigma}.h(e).h(g_1)$ is an execution and reaches the same state as the execution $\tilde{\sigma}.h(g_1).h(e)$. Iterating this argument l times one can show that, similarly, $\tilde{\sigma}.h(e).h(g_1) \ldots h(g_l)$ is an execution and reaches the same state as the execution $\tilde{\sigma}.h(g_1) \ldots h(g_l).h(e)$. This has shown that σ is indeed an execution.

The lemma also requires to prove that any two executions in $\mathit{inter}(C)$ reach the same state. This is straightforward to show using the arguments we have introduced above. We have already shown that any linearization of all events in C is h-labelled by an execution of M that reaches the same state as the execution that labels any other linearization of the same events that fires e last in the sequence. Using this fact and the induction hypothesis it is very simple to complete the proof. ◂

▸ **Lemma 17.** *For any set F of unfolding prefixes, $union(F)$ is the least-upper bound of F with respect to the order $\trianglelefteq$.*

Proof. Let $F := \bigcup_{1\leq i} \mathcal{P}_i$, where $\mathcal{P}_i := \langle E_i, <_i, \#_i, h_i\rangle$ for $1 \leq i$. Let $\mathcal{P} := union(F)$ be their union, where $\mathcal{P} := \langle E, <, \#, h\rangle$. We need to show that

- (upper bound) $\mathcal{P}_i \trianglelefteq \mathcal{P}$;
- (least element) for any unfolding prefix $\mathcal{P}'$ such that $\mathcal{P}_j \trianglelefteq \mathcal{P}'$ holds for all $1 \leq j$, we have that $\mathcal{P} \trianglelefteq \mathcal{P}'$.

We start showing that $\mathcal{P}$ is an upper bound. Let $\mathcal{P}_i \in F$ be an arbitrary unfolding prefix. We show that $\mathcal{P}_i \trianglelefteq \mathcal{P}$:

- Trivially $E_i \subseteq E$.
- $<_i \subseteq < \cap (E_i \times E_i)$. Trivial.
- $<_i \supseteq < \cap (E_i \times E_i)$. Assume that $e < e'$ and that both e and e' are in E_i. Then there is some $1 \leq j$ such that $e <_j e'$, and both e and e' are in E_j. Assume that $e' := \langle t, H\rangle$. Since $\mathcal{P}_j$ is a finite prefix constructed by Def. 3, then necessarily $e \in H$. As a result, Def. 3 must have found that e was in H when adding e' to the prefix that eventually became $\mathcal{P}_i$, and consequently $e' <_i e$.
- $\#_i \subseteq \# \cap (E_i \times E_i)$. Trivial.
- $\#_i \supseteq \# \cap (E_i \times E_i)$. Assume that $e \# e'$ and that $e, e' \in E_i$. We need to prove that $e \#_i e'$. Assume w.l.o.g. that e' was added to $\mathcal{P}_i$ by Def. 3 after e. If e and e' satisfy (8), then trivially $e \#_i e'$. If not, then assume w.l.o.g. that there exists some $e'' < e'$ such that $e \# e''$, and such that e and e'' satisfy (8). Then $e \#_i e''$ and, since $\mathcal{P}_i$ is a LES then we have $e \#_i e'$.
- $h_i = h \cap (E_i \times E_i)$. Trivial.

We now focus on proving that $\mathcal{P}$ is the least element among the upper bounds of F. Let $\mathcal{P}' := \langle E', <', \#', h'\rangle$ be an upper bound of all elements of F. We show that $\mathcal{P} \trianglelefteq \mathcal{P}'$.

- Since E is the union of all E_i and all E_i are by hypothesis in E', then necessarily $E \subseteq E'$.
- $< \subseteq <' \cap (E \times E)$. Assume that $e < e'$. By definition e and e' are in E, so we only need to show that $e <' e'$. We know that there is some $1 \leq i$ such that $e <_i e'$. We also know that $\mathcal{P}_i \trianglelefteq \mathcal{P}'$, which implies that $e <' e'$.
- $< \supseteq <' \cap (E \times E)$. Assume that $e < e'$ and that $e, e' \in E$. We know that there is some $1 \leq i$ such that $e, e' \in E_i$. We also know that $\mathcal{P}_i \trianglelefteq \mathcal{P}'$, which implies that $<_i = <' \cap (E_i \times E_i)$. This means that $e <_i e'$, and so $e < e'$.
- $h = h' \cap (E \times E)$. Trivial.
- $\# \subseteq \#' \cap (E \times E)$. Assume that $e \# e'$. Then e and e' are in E. Two things are possible. Either e, e' satisfy (8) or, w.l.o.g., there exists some $e'' < e'$ such that e and e'' satisfy (8). In the former case, using items above, it is trivial to show that $\neg(e <' e')$, that $\neg(e' <' e)$, and that $h'(e) \diamond\!\!\!\!\diamond h'(e')$. This means that $e \#' e'$. In the latter case its the same.
- $\# \supseteq \#' \cap (E \times E)$. Trivial.

◂

▸ **Theorem 5.** *The unfolding $\mathcal{U}_{M,\diamond}$ exists and is unique. Furthermore, for any non-empty run σ of M, there exists a unique configuration C of $\mathcal{U}_{M,\diamond}$ such that $\sigma \in inter(C)$.*

Proof. Let F be the set of all, finite or infinite, unfolding prefixes of $\mathcal{U}_{M,\diamond}$. By Def. 13 we have that $\mathcal{U}_{M,\diamond} := union(F)$ is an unfolding prefix. By Lemma 17 we know it is $\trianglelefteq$-maximal and unique.

Observe that for a run that fires no transition, i.e. $\sigma = \varepsilon \in T^*$, we may find the empty configuration $\varnothing$ or the configuration $\{\bot\}$, and in both cases σ is an interleaving of the configuration. Hence the restriction to non-empty runs.

Assume that σ fires at least one transition. The proof is by induction on the length $|\sigma|$ of the run.

Base Case. If σ fires one transition t, then t is enabled at $\tilde{s}$, the initial state of M. Then $\{\bot\}$ is a history for t, as necessarily $state(\{\bot\})$ enables t. This means that $e := \langle t, \{\bot\}\rangle$ is an event of $\mathcal{U}_{M,\diamond}$, and clearly $\sigma \in inter(\{\bot, e\})$. It is easy to see that no other event e' different than e but such that $h(e) = h(e')$ can exist in $\mathcal{U}_{M,\diamond}$ and satisfy that the history $\lceil e' \rceil$ of e' equals the singleton $\{\bot\}$. The representative configuration for σ is therefore unique.

Inductive Step. Consider $\sigma = \sigma'.t_{k+1}$, with $\sigma' = t_1.t_2 \dots t_k$. By the induction hypothesis, we assume that there exist a unique configuration C' such that $\sigma' \in inter(C')$. By Lemma 16, all runs in $inter(C')$ reach the same state s and σ' is such a run. Hence, t_{k+1} is enabled at state s. If all $<$-maximal events $e \in \max(C') : h(e)$ interfere with t_{k+1}, then C' is a valid configuration H and by construction (second condition of Def. 3) there is a configuration $C = C' \cup \{e'\}$ with $e' = \langle t_{k+1}, H\rangle$. Otherwise, we construct a valid H by considering sub-configurations of C' removing a maximal event $e \in \max(C') : h(e)$ does not interfere with t_{k+1}. We always reach a valid H since C' is a finite set and $\{\bot\}$ is always a valid H. Considering $C = H \cup \{e'\}$ with $e' = \langle t, H\rangle$, by construction (second condition of Def. 3) we have that $\forall e_H \in H : \neg(e' \# e_H)$ and $\forall e_{\hat{H}} \in C' \setminus H : \neg(e \# e_{\hat{H}})$ (otherwise these events would be in H). Hence, $C' \cup \{e\}$ is a configuration. ◂

B Proofs: Exploration Algorithm

For the rest of this section, as we did in the main sections of the paper, we fix a system $M := \langle \Sigma, T, \tilde{s}\rangle$ and an unconditoinal independence relation $\diamond$ on M. We assume that $reach(M)$ is finite. Let $\mathcal{U}_{\diamond,M} := \langle E, <, \#, h\rangle$ be the unfolding of M under $\diamond$, which we abbreviate as $\mathcal{U}$. For this section, unless otherwise state, we furthermore assume that that $\mathcal{U}$ is finite, i.e., that all computations of M terminate.

Algorithm 1 is recursive, each call to `Explore`(C, D, A) yields either no recursive call, if the function returns at line 4, or one single recursive call (line 9), or two (line 9 and line 11). Furthermore, it is non-deterministic, as e is chosen from either the set $en(C)$ or the set $A \cap en(C)$, which in general are not singletons. As a result, the configurations explored by it may differ from one execution to the next.

For each system M we define the *call graph* explored by Alg. 1 as a directed graph $\langle B, \triangleright\rangle$ representing the actual exploration that the algorithm did on the state space. Different executions will in general yield different call graphs.

The nodes B of the call graph are 4-tuples of the form $\langle C, D, A, e\rangle$, where C, D, A are the parameters of a recursive call made to the funtion `Explore`$(\cdot,\cdot,\cdot)$, and e is the event selected by the algorithm immediately before line 9. More formally, B contains exactly all tuples $\langle C, D, A, e\rangle$ satisfying that

- C, D, and A are sets of events of the unfolding $\mathcal{U}$;
- during the execution of `Explore`$(\varnothing, \varnothing, \varnothing)$, the function `Explore`$(\cdot,\cdot,\cdot)$ has been recursively called with C, D, A as, respectively, first, second, and third argument;
- $e \in E$ is the event selected by `Explore`(C, D, A) immediately before line 9 if C is not maximal; if C is maximal, we define $e := \bot$. [4]

[4] Observe that in this case, if C is maximal, the execution of `Explore`(C, D, A) never reaches line 9.

The edge relation of the call graph, $\triangleright \subseteq B \times B$, represents the recursive calls made by `Explore`$(\cdot,\cdot,\cdot)$. Formally, it is the union of two disjoint relations $\triangleright := \triangleright_l \uplus \triangleright_r$, defined as follows. We define that

$$\langle C, D, A, e\rangle \triangleright_l \langle C', D', A', e'\rangle \quad \text{and that} \quad \langle C, D, A, e\rangle \triangleright_r \langle C'', D'', A'', e''\rangle$$

iff the execution of `Explore`(C, D, A) issues a recursive call to, resp., `Explore`(C', D', A') at line 9 and `Explore`(C'', D'', A'') at line 11. Observe that C' and C'' will necessarily be different (as $C' = C \cup \{e\}$, where $e \notin C$, and $C'' = C$), and therefore the two relations are disjoint sets. We distinguish the node

$$b_0 := \langle \{\bot\}, \varnothing, \varnothing, \bot\rangle$$

as the *initial node*, also called the *root node*. Observe that $\langle B, \triangleright\rangle$ is by definition a weakly connected digraph, as there is a path from the node b_0 to every other node in B. Later in this section we will additionally prove that the call graph is actually a binary tree, where $\triangleright_l$ is the *left-child* relation and $\triangleright_r$ is the *right child* relation.

B.1 General Lemmas

▸ **Lemma 18.** *Let $\langle C, D, A, e\rangle \in B$ be a state of the call graph. We have that*

- *event e is such that $e \in en(C)$;* (9)
- *C is a configuration;* (10)
- *$C \cup A$ is a configuration and $C \cap A = \varnothing$;* (11)
- *$D \subseteq ex(C)$;* (12)
- *if $A = \varnothing$, then $D \subseteq cex(C)$;* (13)
- *for all $e' \in D$ there is some $e'' \in C \cup A$ such that $e' \#^i e''$* (14)

Proof. To show (9) is immediate. Observe, in Alg. 1, that both branches of the "*if*" statement where e is picked select it from the set $en(C)$.

All remaining items, (10) to (14), will be shown by induction on the length $n \geq 0$ of any path

$$b_0 \triangleright b_1 \triangleright \ldots \triangleright b_{n-1} \triangleright b_n$$

on the call graph, starting from the initial node and leading to $b_n := \langle C, D, A, e\rangle$ (we will later show, Lemma 24, that there is actually only one such path). For $i \in \{0, \ldots, n\}$ we define $\langle C_i, D_i, A_i, e_i\rangle := b_i$.

We start showing (10). *Base case.* $n = 0$ and $C = \{\bot\}$. The set $\{\bot\}$ is a configuration. *Step.* Assume C_{n-1} is a configuration. If $b_{n-1} \triangleright_l b_n$, then $C = C_{n-1} \cup \{e\}$ for some event $e \in en(C)$, as stated in (9). By definition, C is a configuration. If $b_{n-1} \triangleright_r b_n$, then $C = C_{n-1}$. In any case C is a configuration.

We show (11), also by induction on n. *Base case.* $n = 0$. Then $C = \{\bot\}$ and $A = \varnothing$. Clearly $C \cup A$ is a configuration and $C \cup A = \varnothing$. *Step.* Assume that $C_{n-1} \cup A_{n-1}$ is a configuration and that $C_{n-1} \cap A_{n-1} = \varnothing$. We have two cases.

- Assume that $b_{n-1} \triangleright_l b_n$. If A_{n-1} is empty, then A is empty as well. Clearly $C \cup A$ is a configuration and $C \cap A$ is empty. If A_{n-1} is not empty, then $C = C_{n-1} \cup \{e\}$ and $A = A_{n-1} \setminus \{e\}$, for some $e \in A_{n-1}$, and we have

  $$C \cup A = (C_{n-1} \cup \{e\}) \cup (A_{n-1} \setminus \{e\}) = C_{n-1} \cup A_{n-1},$$

  so $C \cup A$ is a configuration as well. We also have that $C \cap A = C_{n-1} \cap A_{n-1}$ (recall that $e \notin C$), so $C \cap A$ is empty.

- Assume that $b_{n-1} \rhd_r b_n$ holds. Then we have $C = C_{n-1}$ and also $A = J \setminus C_{n-1}$ for some $J \in \texttt{Alt}(C_{n-1}, D \cup \{e\})$. From (5) we know that $C_{n-1} \cup J$ is a configuration. As a result,

$$C \cup A = C_{n-1} \cup (J \setminus C_{n-1}) = C_{n-1} \cup J,$$

  and therefore $C \cup A$ is a configuration. Finally, by construction of A, we clearly have $C \cap A = \varnothing$.

We show (12), again, by induction on n. *Base case.* $n = 0$ and $D = \varnothing$. Then (12) clearly holds. *Step.* Assume that (12) holds for $\langle C_i, D_i, A_i, e_i \rangle$, with $i \in \{0, \ldots, n-1\}$. We show that it holds for b_n. As before, we have two cases.

- Assume that $b_{n-1} \rhd_l b_n$. We have that $D = D_{n-1}$ and that $C = C_{n-1} \cup \{e_{n-1}\}$. We need to show that for all $e' \in D$ we have $\lceil e' \rceil \subseteq C$ and $e' \notin C$. By induction hypothesis we know that $D = D_{n-1} \subseteq ex(C_{n-1})$, so clearly $\lceil e' \rceil \subseteq C_{n-1} \subseteq C$. We also have that $e' \notin C_{n-1}$, so we only need to check that $e' \neq e_{n-1}$. By contradiction, if $e' = e_{n-1}$, by (14) we would have that some event in C is conflict with some other event in $C \cup A$, which is a contradiction to (11).
- Assume that $b_{n-1} \rhd_r b_n$. We have that $D = D_{n-1} \cup \{e_{n-1}\}$, and by hypothesis we know that $D_{n-1} \subseteq ex(C_{n-1}) = ex(C)$. As for e_{n-1}, by (9) we know that $e_{n-1} \in en(C_{n-1}) = en(C) \subseteq ex(C)$. As a result, $D \subseteq ex(C)$.

We show (13). By (12) we know that $D \subseteq ex(C)$. Assume $A = \varnothing$. For each $e' \in D$ we need to prove the existence of some $e'' \in C$ with $e' \#^i e''$. This is exactly what (14) states.

We show (14), again, by induction on n. *Base case.* $n = 0$ and $D = \varnothing$. The result holds. *Step.* Assume (14) holds for $\langle C_{n-1}, D_{n-1}, A_{n-1}, e_{n-1} \rangle$. We show that it holds for b_n. We distinguish two cases.

- $b_{n-1} \rhd_l b_n$. Then $D = D_{n-1}$. As a result, for any $e' \in D$ there is some $e'' \in C_{n-1} \cup A_{n-1}$ satisfying $e' \#^i e''$. But we have that $C_{n-1} \cup A_{n-1} \subseteq C \cup A$, so such e' is also contained in $C \cup A$, which shows the result.
- $b_{n-1} \rhd_r b_n$. Observe that $D = D_{n-1} \cup \{e_{n-1}\}$. Let $J \in \texttt{Alt}(C_{n-1}, D \cup \{e\})$ be the alternative used to construct $A = J \setminus C_{n-1}$. By definition (6) we know that for all $e' \in D \setminus cex(C_{n-1})$ we can find some $e'' \in J$ with $e' \#^i e''$. We only need to show that $J \subseteq A \cup C$. Observe that this will complete the proof, since for each $e' \in D \cap cex(C_{n-1})$ we already know that there is some $e'' \in C_{n-1} \subseteq C \cup A$ with $e' \#^i e''$. Now, that $J \subseteq C \cup A$ is obvious: $C \cup A = C_{n-1} \cup J \setminus C_{n-1} = C_{n-1} \cup J$.

◂

The following lemma essentially guarantees that whenever Alg. 1 reaches line 8, the set from which e is chosen is not empty.

▸ **Lemma 19.** *If $C \subseteq C'$ are two finite configurations, then $en(C) \cap (C' \setminus C) = \varnothing$ iff $C' \setminus C = \varnothing$.*

Proof. If there is some $e \in en(C) \cap (C' \setminus C)$, then $e \notin C$ and $e \in C'$, so $C' \setminus C$ is not empty. If there is some $e' \in C' \setminus C$, then there is some e'' event that is $<$-minimal in $C' \setminus C$. As a result, $\lceil e'' \rceil \subseteq C$. Since $e'' \notin C$ and $C \cup \{e''\}$ is a configuration (as $C \cup \{e''\} \subseteq C'$), we have that $e'' \in en(C)$. Then $en(C) \cap (C' \setminus C)$ is not empty. ◂

▸ **Lemma 20.** *For any node $\langle C, D, A, e \rangle \in N$ of the call graph we have that $A \neq \varnothing$ implies $en(C) \cap A \neq \varnothing$.*

Proof. The result is a consequence of Lemma 19 and (11). Since $C \cup A$ is configuration that includes C, and $(C \cup A) \setminus C = A$ is not empty, then $en(C) \cap A$ is not empty. ◂

▸ **Lemma 21.** *Let* $b \coloneqq \langle C, D, A, e\rangle$ *and* $b' \coloneqq \langle C', D', A', e'\rangle$ *be two nodes of the call graph such that* $b \rhd b'$*. Then*

- $C \subseteq C'$ *and* $D \subseteq D'$*;* (15)
- *if* $b \rhd_l b'$*, then* $C \subsetneq C'$*;* (16)
- *if* $b \rhd_r b'$*, then* $D \subsetneq D'$*.* (17)

Proof. If $b \rhd_l b'$, then $C' = C \cup \{e\}$ and $D' = D$. Then all the three statements hold. If $b \rhd_r b'$, then $C' = C$ and $D' = D \cup \{e\}$. Similarly, all the three statements hold. ◂

B.2 Termination

▸ **Lemma 22.** *Any path* $b_0 \rhd b_1 \rhd b_2 \rhd \dots$ *in the call graph starting from* b_0 *is finite.*

Proof. By contradiction. Assume that $b_0 \rhd b_1 \rhd \dots$ is an infinite path in the call graph. For $0 \le i$, let $\langle C_i, D_i, A_i, e_i\rangle \coloneqq b_i$. Recall that $\mathcal{U}$ has finitely many events, finitely many finite configurations, and no infinite configuration. Now, observe that the number of times that C_i and C_{i+1} are related by $\rhd_l$ rather than $\rhd_r$ is finite, since every time `Explore(·,·,·)` makes a recursive call at line 9 it adds one event to C_i, as stated by (16). More formally, the set

$$L \coloneqq \{i \in \mathbb{N} : C_i \rhd_l C_{i+1}\}$$

is finite. As a result it has a maximum, and its successor $k \coloneqq 1 + \max_< L$ is an index in the path such that for all $i \ge k$ we have $C_i \rhd_r C_{i+1}$, i.e., the function only makes recursive calls at line 11. We then have that $C_i = C_k$, for $i \ge k$, and by (12), that $D_i \subseteq ex(C_k)$. Recall that $ex(C_k)$ is finite. Observe that, as a result of (16), the sequence

$$D_k \subsetneq D_{k+1} \subsetneq D_{k+2} \subsetneq \dots$$

is an infinite increasing sequence. This is a contradiction, as for sufficiently large $j \ge 0$ we will have that D_{k+j} will be larger than $ex(C_k)$, yet $D_{k+j} \subseteq ex(C_k)$. ◂

▸ **Corollary 23.** *The call graph is a finite directed acyclic graph.*

Proof. Recall that every node $b \in B$ is reachable from the initial node b_0 by definition of the graph. Also, by Lemma 22, all paths from b_0 are finite, and every node has between 0 and 2 adjacent nodes.

By contradiction, if the graph had infinitely may nodes, then König's lemma would guarantee the existence of an infinite path starting from b_0, a contradiction to Lemma 22. Then B is necessarily finite.

As for the acyclicity, again by contradiction, assume that $\langle B, \rhd\rangle$ has a cycle. Then every state of any such cycle would be reachable from b_0, which guarantees the existence of at least one infinite path in the graph. Again, this is a contradiction to Lemma 22. ◂

▸ **Theorem 9** (Termination). *Regardless of its input, Alg. 1 always stops.*

Proof. Remark that Alg. 1 makes calls to three functions, namely, `Extend(·)`, `Remove(·)`, and `Alt(·,·)`, Clearly the first two terminate. Since we gave no algorithm to compute `Alt(·)`, we will assume we employ one that terminates on every input.

Now, observe that there is no loop in Alg. 1. Thus any non-terminating execution of Alg. 1 must perform a non-terminating sequence of recursive calls, which entails the existence of an infinite path in the call graph associated to the execution. Since, by Lemma 22, no infinite path exist in the call graph, Alg. 1 always terminates. ◂

B.3 Optimality

▸ **Lemma 24.** *Let $b, b_1, b_2, b_3, b_4 \in B$ be nodes of the call graph such that*

$$b \rhd_l b_1 \rhd^* b_3 \quad \text{and} \quad b \rhd_r b_2 \rhd^* b_4.$$

and such that $\langle C_3, D_3, A_3, e_3\rangle := b_3$ and $\langle C_4, D_4, A_4, e_4\rangle := b_4$. Then $C_3 \neq C_4$.

Proof. Let $\langle C, D, A, e\rangle := b$, $\langle C_1, D_1, A_1, e_1\rangle := b_1$, and $\langle C_2, D_2, A_2, e_2\rangle := b_2$. By (16) we know that $e \in C_1$, and by (15) that $e \in C_3$. We show that $e \notin C_4$. By (17) we have that $e \in D_2$, and again by (15) that $e \in D_4$. Since $D_4 \subseteq ex(C_4)$, by (12), we have that $e \in ex(C_4)$, so $e \notin C_4$. ◂

▸ **Corollary 25.** *The call graph $(B, \rhd)$ is a finite binary tree, where $\rhd_l$ and $\rhd_r$ are respectively the* left-child *and* right-child *relations.*

Proof. Corollary 23 states that the call graph is a finite directed acyclic graph. Lemma 24 guarantees that for every node $b \in B$, the nodes reached after the left child are different from those reached after the right one. ◂

▸ **Lemma 26.** *For any maximal configuration $C \subseteq E$, there is* at most one *node $\langle \tilde{C}, \tilde{D}, \tilde{A}, \tilde{e}\rangle \in B$ with $C = \tilde{C}$.*

Proof. By contradiction, assume there was two different nodes,

$$\hat{b} := \langle C, \hat{D}, \hat{A}, \hat{e}\rangle \quad \text{and} \quad b' := \langle C, D', A', e'\rangle$$

in B such that the first component of the tuple is C. The call graph is a binary tree, because of Corollary 25, so there is exactly one path from $b_0 := \langle \varnothing, \varnothing, \varnothing, e_0\rangle$ to respectively $\hat{b}$ and b'. Let

$$\hat{b}_0 \rhd \hat{b}_1 \rhd \ldots \rhd \hat{b}_{n-1} \rhd \hat{b}_n \quad \text{and} \quad b'_0 \rhd b'_1 \rhd \ldots \rhd b'_{m-1} \rhd b'_m$$

be the two such unique paths, with $\hat{b}_n := \hat{b}$, $b'_n := b'$ and $\hat{b}_0 := b'_0 := b_0$. Such paths clearly share the first node b_0. In general they will share a number of nodes to later diverge. Let i be the index of the last node common to both paths, i.e., the maximum integer $i \geq 0$ such that

$$\langle \hat{b}_0, \hat{b}_1, \ldots, \hat{b}_i\rangle = \langle b'_0, b'_1, \ldots, b'_i\rangle$$

holds. Observe both paths necessarily diverge before reaching the last node, i.e., one cannot be a prefix of the other. This is because both $\hat{b}$ and b' are leaves of the call graph, i.e., there is no $b'' \in B$ such that either $\hat{b} \rhd b''$ or $b' \rhd b''$. As a result $\hat{b} \neq b'_j$ for any $j \in \{0, \ldots, m\}$ and $b' \neq \hat{b}_j$ for any $j \in \{0, \ldots, n\}$. This means that $i < \min\{n, m\}$.

Let $\langle C_i, D_i, A_i, e_i\rangle := b_i$. W.l.o.g., assume that $\hat{b}_i \rhd_l \hat{b}_{i+1}$ and that $b'_i \rhd_r b'_{i+1}$. Now, using (16) and (15), it is simple to show that $e_i \in C$. And using (17) and (15), that $e_i \in D'$. Then, by (12) we get that $e_i \in ex(C)$, a contradiction to $e_i \in C$. ◂

▸ **Theorem 10** (Optimality). *Let $\tilde{C}$ be a maximal configuration of $\mathcal{U}$. Then* `Explore(·,·,·)` *is called* at most once *with its first parameter being equal to $\tilde{C}$.*

Proof. By construction, every call to `Explore(`C, D, A`)` produces one node of the form $\langle C, D, A, e\rangle$, for some $e \in E$, in the call graph associated to the execution. By Lemma 26,

there is at most one node with its first parameter being equal to $\tilde{C}$, so $\texttt{Explore}(\cdot,\cdot,\cdot)$ can have been called at most once with $\tilde{C}$ as first parameter.

Observe, furthermore, that the algorithm does not initiate what Abdulla et al. call *sleep-set blocked executions* [2]. These correspond, in our setting, to exploring the same configuration in both branches of the tree. Formally, our algorithm would explore sleep-set blocked executions iff it is possible to find some $b \in B$ such that the left and right subtrees of b contain nodes exploring the same configuration. By Lemma 24 this is not possible. ◂

B.4 Completeness

▸ **Lemma 27.** *Let $b := \langle C, D, A, e\rangle \in B$ be a node in the call graph and $\hat{C} \subseteq E$ an arbitrary maximal configuration of $\mathcal{U}$ such that $C \subseteq \hat{C}$ and $D \cap \hat{C} = \varnothing$. Then exactly one of the following statements hold:*

- *Either C is a maximal configuration of $\mathcal{U}$, or*
- *$e \in \hat{C}$ and b has a left child, or*
- *$e \notin \hat{C}$ and b has a right child.*

Proof. The proof is by induction on b using a specific total order in B that we define now. Recall that $\langle B, \rhd\rangle$ is a binary tree (Corollary 25). We let $\lessdot \subseteq B \times B$ be the unique in-order relation in B. Formally, $\lessdot$ is the order that sorts, for every $\tilde{b} \in B$, first all nodes reachable from $\tilde{b}$'s left child (if there is any), then $\tilde{b}$, then all nodes reachable from $\tilde{b}$'s right child (if there is any).

Base case. Node b is the least element in B w.r.t. $\lessdot$. Then b is the leftmost leaf of the call tree, i.e., $b_0 \rhd_l^* b$, and C is a maximal configuration. Then the first item holds.

Step case. Assume that the result holds for any node $\tilde{b} \lessdot b$. If C is maximal, we are done. So assume that C is not maximal, and so that b has at least one left child. If $e \in \hat{C}$, then we are done, as the second item holds.

So assume that $e \notin \hat{C}$. The rest of this proof shows that the third item of the lemma holds, i.e., that b has right child. In particular we show that there exists some alternative $\hat{J} \subseteq \hat{C}$ such that $\hat{J} \in \texttt{Alt}(C, D \cup \{e\})$.

We start by setting up some notation. Observe that any alternative $J \in \texttt{Alt}(C, D \cup \{e\})$ needs to contain, for every event $e' \in D \cup \{e\}$, some event $e'' \in J \cup C$ in immediate conflict with e', cf. (6). In fact e'' can be in J or in C. Those $e' \in D \cup \{e\}$ such that C already contains some e'' in conflict with e' pose no problem. So we need to focus on the remaining ones, we assign them a specific name, we define the set

$$F := \{e_1, \ldots, e_n\} := D \setminus \mathit{cex}(C) \cup \{e\}.$$

Let e_i be any event in F. Clearly $e_i \in \mathit{cex}(\hat{C})$, as $e_i \in D \subseteq \mathit{ex}(C)$, by (12), and so $\lceil e_i \rceil \subseteq C \subseteq \hat{C}$ and $e_i \notin \hat{C}$. Since $e_i \in \mathit{cex}(\hat{C})$ we can find some $e_i' \in \hat{C}$ such that $e_i \#^i e_i'$. We can now define a set

$$\hat{J} := [\{e_1', \ldots, e_n'\}]$$

such that $e_i' \in \hat{C}$ and $e_i \#^i e_i'$ for $i \in \{1, \ldots, n\}$. Clearly $\hat{J} \subseteq \hat{C}$ and $\hat{J}$ is causally closed, so it is a configuration. Observe that $\hat{J}$ is not uniquely defined, there may be several e_i' to choose for each e_i (some of the e_i' might even be the same). We take any e_i' in immediate conflict with e_i, the choice is irrelevant (for now).

We show now that $\hat{J} \in \texttt{Alt}(C, D \cup \{e\})$ when function $\texttt{Alt}(\cdot)$ is called just before line 11 during the execution of $\texttt{Explore}(C, D, A)$. Let $\hat{U}$ be the set of events contained in variable U

of Alg. 1 exactly when `Alt(·)` is called. Clearly $C \cup \hat{J}$ is a configuration, so (6) holds. To verify (5), consider any event $\tilde{e} \in D \cup \{e\}$. If $\tilde{e} \in D \cap cex(C)$ we can always find some $\tilde{e}' \in C$ with $\tilde{e}' \in \#^i_{\hat{U}}(\tilde{e})$. If not, then $\tilde{e} = e_i$ for some $i \in \{1, \ldots, n\}$ and we can find some $e'_i \in \hat{J}$ such that $e_i \#^i e'_i$. In order to verify (5) we only need to check that $e'_i \in \hat{U}$. In the rest of this proof we show this. Observe that $e'_i \in \hat{U}$ also implies that $\hat{J} \subseteq \hat{U}$, necessary to ensure that $\hat{J}$ is an alternative to $D \cup \{e\}$ after C *when the function* `Alt(·)` *is called*.

In the sequel we show that $\hat{J} \subseteq \hat{U}$. In other words, that event e'_i, for $i \in \{1, \ldots, n\}$, is present in set U when function `Alt`$(C, D \cup \{e\})$ is called. The set U has been filled with events in function `Extend(·)` as the exploration of $\mathcal{U}$ advanced, some of them have been kept in U, some of them have been removed with `Remove(·)`. To reason about the events in $\hat{U}$ we need to look at the fragment of $\mathcal{U}$ explored so far.

For $i \in \{1, \ldots, n\}$ let $b_i := \langle C_i, D_i, A_i, e_i \rangle \in B$ be the node in the call graph associated to event $e_i \in F$. These nodes are all situated in the unique path from b_0 to b. W.l.o.g. assume (after possible reordering of the index i) that

$$b_0 \rhd^* b_1 \rhd^* b_2 \rhd^* \ldots \rhd^* b_n$$

where $b_n = b$ and $e_n = e$. First observe that for any $i \in \{2, \ldots, n\}$ we have $\{e_1, \ldots, e_{i-1}\} \subseteq D_i$. Since every event e_i is in $D = D_n$, for $i \in \{1, \ldots, n-1\}$, we know that the first step in the path that goes from b_i to b_{i+1} is a *right child*. In other words, the call to `Explore`(C_i, D_i, A_i) is right now blocked on the *right-hand side recursive call* at line 11 in Alg. 1, after having decided that there was one right child to explore. For the shake of clarity, we can then informally write

$$b_0 \rhd^* b_1 \rhd_r \rhd^* b_2 \rhd_r \rhd^* \ldots \rhd_r \rhd^* b_n.$$

We additionally define the sets of events

$$U_0, U_1, \ldots, U_n \subseteq E$$

as, respectively for $i \in \{1, \ldots, n\}$, the value of the variable U during the execution of `Explore`(C_i, D_i, A_i) just before the *right recursive call* at line 11 was made, i.e., the value of variable U when `Alt`$(C_i, D_i \cup \{e_i\})$ was called. For $i = 0$ we set $U_0 := \{\bot\}$ to the initial value of U. According to this definition we have that $U_n = \hat{U}$.

To prove that $\hat{J} \subseteq \hat{U} = U_n$ it is now sufficient to prove that $e'_i \in U_i$, for $i \in \{1, \ldots, n\}$. This is essentially because of the following three facts.

1. Clearly $e_i \in U_i$.
2. For any node $\tilde{b} := \langle \tilde{C}, \tilde{D}, \cdot, \tilde{e} \rangle \in B$ explored after b_i and before b_n it holds that $e_i \in \tilde{D}$, by (15), and so every time function `Remove`$(\tilde{e}, \tilde{C}, \tilde{D})$ has been called, event e_i has not been removed from U.
3. Any event in immediate conflict with e_i will likewise not be removed from set U as long as e_i remains in D, for the same reason as before.

In other words, $e'_i \in U_i$ implies that $e'_i \in U_n$, for $i \in \{1, \ldots, n\}$.

We need to show that $e'_i \in U_i$, for $i \in \{1, \ldots, n\}$. Consider the configuration $C' \subseteq E$ defined as follows:

$$C' := C \cup \{e_i\} \cup \lceil e'_i \rceil.$$

First, note that C' is indeed a configuration, since it is clearly causally closed and there is no conflict: $e_i \in en(C)$ and $C \cup \lceil e'_i \rceil \subseteq \hat{C}$ and $\lceil e_i \rceil \cup \lceil e'_i \rceil$ is conflict-free (because e_i and e'_i are in *immediate* conflict). Remark also that $D_i \subseteq ex(C')$ (this is because $D \cap \hat{C} = \varnothing$) and that $e'_i \in cex(C')$. We now consider two cases:

- *Case 1*: there is some maximal configuration $C'' \supseteq C'$ such that $D_i \cap C'' = \varnothing$. We show that C'' have been visited during the exploration of the left subtree of b_i. In that case, since $e_i' \in cex(C'')$ and $e_i \in C''$, Alg. 1 will have been appended e_i' to U during that exploration, and e_i' will remain in U at least as long as e_i is in D.
  To show that C'' has been explored, consider the left child $b_i' := \langle C_i \cup \{e_i\}, D_i, \cdot, \cdot \rangle$ of b_i. In that case, since $b_i \lessdot b$ (recall that b is in the right subtree of b_i), clearly every node $\hat{b} \in B$ in the subtree rooted at b_i' (i.e., $b_i' \rhd^* \hat{b}$) is such that $\hat{b} \lessdot b_i \lessdot b$. This means that the induction hypothesis applies to $\hat{b}$. So Lemma 28 applied to b_i' and C'' shows that C'' has been explored in the subtree rooted at b_i'. As a result $e_i' \in U_i$ and $e_i' \in U_n$, what we wanted to prove.
- *Case 2*: there is no maximal configuration $C'' \supseteq C'$ such that $D_i \cap C'' = \varnothing$. In other words, *any* maximal configuration $C'' \supseteq C'$ is such that $D_i \cap C'' \neq \varnothing$. Our first step is showing that this implies that

$$\exists j \in \{1, \ldots, i-1\} \text{ such that } \#(e_i) \cap \hat{C} \supseteq \#(e_j) \cap \hat{C}. \tag{18}$$

  Let $C'' \supseteq C'$ be a maximal configuration. Then $D_i \cap C'' \neq \varnothing$. This implies that $D_i \cap en(C) \cap C'' \neq \varnothing$, as necessarily $D_i \cap C'' \subseteq en(C)$. Observe that $D_i \cap en(C) = \{e_1, \ldots, e_{i-1}\}$, so we have that $\{e_1, \ldots, e_{i-1}\} \cap C'' \neq \varnothing$. Consider now the following two sets:

$$X_1 := \hat{C} \setminus \#(e_i) \quad \text{and} \quad X_2 := X_1 \cup \{e_i\}.$$

  Observe now the following. We can find a maximal configuration $C''' \supseteq X_1$ satisfying that $D_i \cap C''' = \varnothing$ (for instance, take $C''' := \hat{C}$). But, because $C' \subseteq X_2$, we cannot find any $C''' \supseteq X_2$ satisfying that $D_i \cap C''' = \varnothing$. This implies that for any $C''' \supseteq X_2$ we have $\{e_1, \ldots, e_{i-1}\} \cap C''' \neq \varnothing$. Based on the last statement we can now prove (18) by contradiction. Assume that (18) does not hold. Then for any $j \in \{1, \ldots, i-1\}$, one could find some event $\tilde{e} \in \#(e_j) \cap \hat{C}$ such that $\tilde{e} \notin \#(e_i) \cap \hat{C}$. Then $\tilde{e} \notin \#(e_i)$ and as a result $\tilde{e} \in X_1 \subseteq X_2$. This now would mean that for any $j \in \{1, \ldots, i-1\}$ it holds that $\#(e_j) \cap X_2 \neq \varnothing$. This implies that *any* maximal configuration C''' extending X_2 is such that $\{e_1, \ldots, e_{i-1}\} \cap C''' = \varnothing$. This is a contradiction, so the validity of (18) is now established.
  According to (18) there might be several integers $j \in \{1, \ldots, i-1\}$ such that $\#(e_i) \cap \hat{C} \supseteq \#(e_j) \cap \hat{C}$ holds. Let m be the minimum such j, and consider the following set:

$$X_3 := X_1 \cup \{e_m\} \cup \lceil e_m' \rceil.$$

  We will now prove that X_3 is a configuration and it has been visited during the exploration of the subtree rooted at the left child of b_m. We first establish several claims about X_3:
  - *Fact 1: set X_3 is causally closed.* Since X_1 is causally closed, clearly $X_1 \cup \lceil e_m' \rceil$ is causally closed. Now, since $\{e_i, e_m\} \subseteq en(C)$, we have that $\#(e_i) \cap C = \varnothing$, and as a result $\lceil e_m \rceil \subseteq C \subseteq X_1 \subseteq X_3$.
  - *Fact 2: set X_3 is conflict free.* Since $X_1 \cup \lceil e_m' \rceil \subseteq \hat{C}$, there is no pair of confliting events in $X_1 \cup \lceil e_m' \rceil$. Consider now e_m. Since e_m and e_m' are in *immediate* conflict, by definition e_m has no conflicth with any event in $\lceil e_m' \rceil$. Consider now any event $\tilde{e} \in X_1$. Observe that $\tilde{e} \in \hat{C}$. If $\tilde{e} \in \#(e_m)$, then by (18) we have that $\tilde{e} \in \#(e_i)$, which implies that $\tilde{e} \notin X_1$. So e_m has no conflict with any event in X_1.
  - *Fact 3: it holds that $C_m \cup \{e_m\} \subseteq X_3$.* Since $C_m \subseteq C$, by (16), and $C \subseteq X_1 \subseteq X_3$, we clearly have that $C_m \subseteq X_3$. Also, $e_m \in X_3$ by definition.

- *Fact 4: it holds that* $X_3 \cap D_m = \varnothing$. By (12) and (15) we know that $D_m \subseteq D \subseteq ex(C)$. Since the sets $en(C)$ and $cex(C)$ partition $ex(C)$ we make the following argument. For any $\tilde{e} \in D_m \cap cex(C)$ we know that $\tilde{e} \notin X_3$, as $C \subseteq X_3$. As for $D_m \cap en(C)$ we have that $D_m \cap en(C) = \{e_1, \ldots, e_{m-1}\}$. So for any $j \in \{1, \ldots, m-1\}$, because of the minimality of m, we know that $\#(e_i) \cap \hat{C} \supseteq \#(e_j) \cap \hat{C}$ does not hold. In other words, we know that there exists at least one event $\tilde{e} \in \#(e_j) \cap \hat{C}$ such that $\tilde{e} \notin \#(e_i) \cap \hat{C}$. This implies that $\tilde{e} \notin \#(e_i)$, and as a result $\tilde{e} \in X_1 \subseteq X_3$. So, for any event in D_m there is at least one conflicting event in X_3, and X_3 is a configuration. Therefore $X_3 \cap D_m = \varnothing$.

To show that X_3 has been explored in the subtree rooted at b_m, consider the left child $b'_m := \langle C_m \cup \{e_m\}, D_m, \cdot, \cdot \rangle$ of b_m. The induction hypothesis applies to any node $\hat{b} \in B$ in the subtree rooted at b'_m (i.e., $b'_m \rhd^* \hat{b}$). This is because $\hat{b} \lessdot b'_m \lessdot b_m \lessdot b$. By the first two facts previously proved, we know that X_3 is a configuration. The last two facts, together with the fact that the induction hypothesis holds on the subtree rooted at b'_m, imply, by Lemma 28, that some maximal configuration $C'' \supseteq X_3$ has been explored in the subtree rooted at b'_m. Since $e_m \in X_3$ and $e'_m \in cex(X_3) \subseteq cex(C'')$, we know that e'_m have been discovered at least when exploring C''. Since $e_m \#^i e'_m$ and e_m is in set D we also know that `Remove(·)` cannot remove e'_m from U before e_m is removed from D. This implies that $e'_m \in U_m$, but also that $e'_m \in U_n$.

Now, our goal was proving that $e'_i \in U_n$. Since $e'_m \in \#(e_i)$, by (18), there is some $\tilde{e} \in \#^i(e_i)$ such that $\tilde{e} \leq e'_m$. Since U_n is causally closed, we have that $\tilde{e} \in U_n$.

We have found some event $\tilde{e} \in U_n$ such that $e_i \#^i \tilde{e}$. If $\tilde{e} \neq e'_i$, then we substitute e'_i in $\hat{J}$ by $\tilde{e}$. This means that in the definition of $\hat{J}$ we cannot chose any arbitrary e'_i from $\hat{C}$ (as we said before, to keep things simple). But we can always find at least one event in $\hat{C}$ that is in immediate conflict with e_i and is also present in U_n. Observe that the choice made for e_i, with $i \in \{1, \ldots, n\}$ has no consequence for the choices made for $j \in \{1, \ldots, i-1\}$. This means that we can always make a choice for index i after having made choices for every $j < i$.

This completes the argument showing that every e'_i (possibly modifying the original choice) is in $\hat{U}$, and shows that $\hat{J} \subseteq \hat{U}$. This implies, by construction of $\hat{J}$, that $\hat{J} \in$ `Alt(`$C, D \cup \{e\}$`)` when the set of events U present in memory equals $\hat{U}$. As a result, Alg. 1 will do a recursive call at line 11 and b will have a right child. This is what we wanted to prove. ◂

▸ **Lemma 28.** *For any node* $b := \langle C, D, \cdot, e \rangle \in B$ *in the call graph and any maximal configuration* $\hat{C} \subseteq E$ *of* $\mathcal{U}$*, if* $C \subseteq \hat{C}$ *and* $D \cap \hat{C} = \varnothing$ *and Lemma 27 holds on all nodes in the subtree rooted at* b*, then there is a node* $b' := \langle C', \cdot, \cdot, \cdot \rangle \in B$ *such that* $b \rhd^* b'$*, and* $\hat{C} = C'$*.*

Proof. Assume that Lemma 27 holds on any node $b'' \in B$ such that $b \rhd^* b''$, i.e., all nodes in the subtree rooted at b. Since $C \subseteq \hat{C}$ and $D \cap \hat{C} = \varnothing$, we can apply Lemma 27 to b and $\hat{C}$. If C is maximal, then clearly $C = \hat{C}$ and we are done. If not we consider two cases. If $e \in \hat{C}$, then by Lemma 27 we know that b has a left child $b_1 := \langle C_1, D_1, \cdot, e_1 \rangle$, with $C_1 := C \cup \{e\}$ and $D_1 := D$. Finally, if $e \notin \hat{C}$, then equally by Lemma 27 we know that b has a right child $b_1 := \langle C_1, D_1, \cdot, e_1 \rangle$, with $C_1 := C$ and $D_1 := D \cup \{e\}$. Observe, in any case, that $C_1 \subseteq \hat{C}$ and $D_1 \cap \hat{C} = \varnothing$.

If C_1 is maximal, then necessarily $C_1 = \hat{C}$, we take $b' := b_1$ and we have finished. If not, we can reapply Lemma 27 at b_1 and make one more step into one of the children b_2 of b_1. If C_2 still not maximal (thus different from $\hat{C}$) we need to repeat the argument starting from b_2 only a finite number n of times until we reach a node $b_n := \langle C_n, D_n, \cdot, \cdot \rangle$ where C_n

is a maximal configuration. This is because every time we repeat the argument on a non-maximal node b_i we advance one step down in the call tree, and all paths in the tree are finite. So eventually we find a leaf node b_n where C_n is maximal and satisfies $C_n \subseteq \hat{C}$. This implies that $C_n = \hat{C}$, and we can take $b' := b_n$. ◂

▸ **Theorem 11** (Completeness). *Let $\tilde{C}$ be a maximal configuration of $\mathcal{U}$. Then* `Explore`$(\cdot,\cdot,\cdot)$ *is called* at least once *with its first parameter being equal to $\tilde{C}$.*

Proof. We need to show that for every maximal configuration $\hat{C} \subseteq E$ we can find a node $b := \langle C, \cdot, \cdot, \cdot \rangle$ in B such that $\hat{C} = C$. This is a direct consequence of Lemma 28. Consider the root node of the tree, $b_0 := \langle C, D, A, \bot \rangle$, where $C = \{\bot\}$ and $D = A = \varnothing$. Clearly $C \subseteq \hat{C}$ and $D \cap \hat{C} = \varnothing$, and Lemma 27 holds on all nodes of the call tree. So Lemma 28 applies to $\hat{C}$ and b_0, and it establishes the existence of the aforementionned node b. ◂

B.5 Memory Consumption

The following proposition establishes that Alg. 1 cleans set U adequately, and that after finishing the execution of `Explore`(C, D, A), set U has the form described by the proposition.

▸ **Proposition 29.** *Assume the function* `Explore`(C, D, A) *is eventually called. Let $\tilde{U}$ and $\hat{U}$ be, respectively, the values of set U in Alg. 1 immediately before and immediately after executing the call. If $Q_{C,D,\tilde{U}} \subseteq \tilde{U} \subseteq Q_{C,D,\tilde{U}} \cup en(C)$, then $\hat{U} = Q_{C,D,\hat{U}}$.*

Proof. Let $b := \langle C, D, A, e \rangle \in B$ be the node in the call tree associated to the call to `Explore`(C, D, A). The proof is by induction on the length of the longest path to a leaf starting from b (in the subtree rooted at b).

Base case. The length is 0, b is leaf node, and C is a maximal configuration. Then $en(C) = \varnothing$, so $\tilde{U} \subseteq Q_{C,D,\tilde{U}}$. By hypothesis $Q_{C,D,\tilde{U}} \subseteq \tilde{U}$ also holds, so $\tilde{U} = Q_{C,D,\tilde{U}}$. Now, the call to `Extend`$(C)$ adds to U only events from $cex(C)$. So at line 4, clearly $\hat{U} = Q_{C,D,\hat{U}}$.

Step case. Let $U_1 := \tilde{U}$ be the value of set U immediately before the call to the function `Explore`(C, D, A). Let U_2 be the value immediately before Alg. 1 makes the first recursive call, at line 9; U_3 the value immediately after that call returns; U_4 immediately after the second recursive call returns; and $U_5 := \hat{U}$ immediately after the call to `Explore`(C, D, A) returns. Assume that $Q_{C,D,U_1} \subseteq U_1 \subseteq Q_{C,D,U_1} \cup en(C)$ holds. Let $C' := C \cup \{e\}$. We first show that

$$Q_{C',D,U_2} \subseteq U_2 \subseteq Q_{C',D,U_2} \cup en(C')$$

holds. This ensures that the induction hypothesis applies to the first recursive call, at line 9, and guarantees that $U_3 = Q_{C',D,U_3}$.

Let $\tilde{e}$ be an event in Q_{C',D,U_2}. We show that $\tilde{e} \in U_2$. First, remark that $U_2 = U_1 \cup ex(C)$. If $\tilde{e} \in C \cup D \subseteq U_1 \subseteq U_2$, we are done. If $\tilde{e} = e$, then clearly $\tilde{e} \in ex(C) \subseteq U_2$. Otherwise $\tilde{e}$ is in $[e_1]$ for some $e_1 \in U_2$ such that there is some $e_2 \in C' \cup D$ with $e_1 \#^i e_2$. Since celarly U_2 is causally closed and $e_1 \in U_2$, we have that $\tilde{e} \in U_2$.

Let $\tilde{e}$ be now an event in U_2. We show that $\tilde{e} \in Q_{C',D,U_2} \cup en(C')$. If $\tilde{e} \in U_1$, the clearly $\tilde{e} \in Q_{C',D,U_2}$ (esentially because $U_1 \subseteq U_2$). So assume that $\tilde{e} \in U_2 \setminus U_1 = ex(C)$. Now, observe that $ex(C) \subseteq \{e\} \cup ex(C')$. We are done if $\tilde{e} \in \{e\} \cup en(C')$, so assume that $\tilde{e} \in cex(C')$. Since $C' \subseteq U_2$ and $\tilde{e} \in U_2$, by definition we have $\tilde{e} \in Q_{C',D,U_2}$. This shows that $\tilde{e} \in Q_{C',D,U_2} \cup en(C')$.

Then by induction hypothesis we have that $U_3 = Q_{C',D,U_3}$ immediately after the recursive call of line 9 returns. Function `Alt`$(\cdot)$ does not update U, so when the second recursive call is made, line 11, clearly

$$Q_{C,D',U_3} \subseteq U_3 \subseteq Q_{C,D',U_3} \cup en(C)$$

holds, with $D' \coloneqq D \cup \{e\}$. This is obvious after realizing the fact that

$$Q_{C\cup\{e\},D,U_3} = Q_{C,D\cup\{e\},U_3}.$$

So the induction hypothesis applies to the second recursive call as well, and guarantees that $U_4 = Q_{C,D\cup\{e\},U_4}$ holds immediately after the recursive call of line 11 returns.

Recall that our goal is proving that $U_5 = Q_{C,D,U_5}$. The difference between U_4 and U_5 are the events removed by the call to the function `Remove`(e, C, D). Let R be such events (see below for a formal definition). Then we have that $U_5 = U_4 \setminus R$. In the sequel we show that the following equalities hold:

$$U_5 = U_4 \setminus R = Q_{C,D\cup\{e\},U_4} \setminus R = Q_{C,D,U_4} = Q_{C,D,U_5} \tag{19}$$

Observe that these equalities prove the lemma. In the rest of this proof we prove the various equalities above.

To prove (19), first observe that the events removed from U by `Remove`(e, C, D), called R above, are exactly

$$R \coloneqq \left(\{e\} \cup \bigcup_{e' \in \#^i_{U_4}(e)} [e']\right) \setminus Q_{C,D,U_4}. \tag{20}$$

This is immediate from the definition of `Remove`$(\cdot)$. Now we prove two statements, (21) and (22), that imply the validity of (19). We start stating the first:

$$Q_{C,D\cup\{e\},U_4} \setminus R = Q_{C,D,U_4}. \tag{21}$$

This equality intuitively says that (left-hand side) executing `Remove`(e, C, D) when the set U contains the events in U_4 (remember that $U_4 = Q_{C,D\cup\{e\},U_4}$) leaves in U exactly (right-hand side) all events in C, all events in D, and all events that causally precede some other event from U (in fact, U_4) which is is conflict with some event in $C \cup D$. For the shake of clarity, unfolding the definitions in (21) yields the following equivalent equality:

$$\left(C \cup D \cup \{e\} \cup \bigcup_{\substack{e' \in C\cup D\cup\{e\} \\ e'' \in \#^i_{U_4}(e')}} [e'']\right) \setminus \left(\left(\{e\} \cup \bigcup_{e' \in \#^i_{U_4}(e)} [e']\right) \setminus Q_{C,D,U_4}\right) = Q_{C,D,U_4}$$

We now prove (21). Let $\tilde{e}$ be an event contained in the left-hand side. We show that $\tilde{e}$ is in Q_{C,D,U_4}. We are done if $\tilde{e} \in C \cup D$. If $\tilde{e} = e$, then $\tilde{e} \notin R$. Now, from the definition (20) of R we get that $\tilde{e} \in Q_{C,D,U_4}$. Lastly, if $\tilde{e} \notin C \cup D \cup \{e\}$, then there is some event $e' \in C \cup D \cup \{e\}$ and some event $e'' \in U_4$ such that $e' \#^i e''$ and $\tilde{e} \leq e''$. If $e' \in C \cup D$, then by defnition $\tilde{e} \in Q_{C,D,U_4}$. The case that $e' = e$ cannot happen, as we show now. Since $\tilde{e}$ is in the left-hand side, $\tilde{e}$ is not in R. If $\tilde{e} \notin R$, then $\tilde{e}$ is either in Q_{C,D,U_4}, as we wanted to show, or $\tilde{e}$ is not in $\{e\} \cup \bigcup_{\hat{e} \in \#^i_{U_4}(e)} [\hat{e}]$. This means that $e' \neq e$.

For the opposite direction, let $\tilde{e}$ be an event in Q_{C,D,U_4}. We show that it is contained in the left-hand side set. By definition $\tilde{e} \notin R$. If $\tilde{e} \in C \cup D$, clearly $\tilde{e}$ is in the left-hand side. If not, then there is some event $e' \in C \cup D$ and some event $e'' \in U_4$ such that $e' \#^i e''$ and $\tilde{e} \leq e''$. Then by definition $\tilde{e}$ is in the left-hand side. This completes the proof of (21).

The second statement necessary to prove (19) is the following:

$$Q_{C,D,U_4} = Q_{C,D,U_5} \tag{22}$$

From left to right. Assume that $\tilde{e} \in Q_{C,D,U_4}$. Routinary if $\tilde{e} \in C \cup D$. Assume otherwise that there is some $e_1 \in C \cup D$ and $e_2 \in \#^i_{U_4}(e_1)$ such that $\tilde{e} \in [e_2]$. We show that $e_2 \in U_5$, which clearly proves that $\tilde{e} \in Q_{C,D,U_5}$. By definition $e_2 \in U_4$. By (20), clearly $e_2 \notin R$, as $e_2 \in Q_{C,D,U_4}$. Since $U_5 = U_4 \setminus R$ we have that $e_2 \in U_5$.

From right to left the proof is even simpler. Assume that $\tilde{e} \in Q_{C,D,U_4}$. Routinary if $\tilde{e} \in C \cup D$. Assume otherwise that there is some $e_1 \in C \cup D$ and $e_2 \in \#^i_{U_5}(e_1)$ such that $\tilde{e} \in [e_2]$. Since $U_5 \subseteq U_4$, clearly $e_2 \in U_4$ and so $e_2 \in Q_{C,D,U_4}$. Then $\tilde{e} \in Q_{C,D,U_4}$ as the latter is causally closed. ◂

C Proofs: Improvements

C.1 Completeness with Cutoffs

In § 5.2 we describe a modified version of Alg. 1, where the `Extend` procedure has been replaced by the `Extend'` procedure. The updated version uses a predicate $\mathsf{cutoff}(e, U, G)$ to decide when an event is added to U. We refer to this version as the *updated algorithm*.

Like Alg. 1, the updated algorithm also explores a binary tree. It works by, intuitively, "*allowing*" Alg. 1 to "*see*" only the non-cutoff events. The terminal configurations it will explore, i.e., those at which the procedure `en`(C) of Alg. 1 returns an empty set, will be those for which any enabled event in $en(C)$ has been declared a cutoff.

Many properties remain true in the updated algorithm, e.g., Lemma 18. Consider the set of terminal configurations explored by the updated algorithm, and let us denote them by

$$C_1, C_2, \ldots, C_n.$$

Let $\mathcal{P}' := \langle E', <', \#' \rangle$ be the unique prefix of $\mathcal{U}$ whose set of events E' equals $\bigcup_{1 \leq i \leq n} C_i$. Whenever Alg. 1 is applied to an acyclic state-space (all executions terminate), the following properties hold:

- $\mathcal{P}' = \mathcal{U}$;
- Each configuration C_i is a maximal configuration of $\mathcal{P}'$.

However, when we apply the updated algorithm to an arbitrary system (with possibly non-terminating executions), none of these properties remain valid in general. Obviously the first one will not be valid, e.g., if $\mathcal{U}$ is infinite, this was expected and intended. The second property will also not be valid in general, essentially because one event could be declared as cutoff when exploring one configuration and as non-cutoff when exploring another configuration. We illustrate this with an abstract example.

▸ **Example 30.** Assume that $\mathcal{U}$ is infinite and has only two maximal (infinite) configurations. The updated algorithm will explore the first until reaching some first terminal (and finite) configuration C_1 where all events in $en(C_1)$ have been declared as cutoffs. Let e be one of those cutoffs in $en(C_1)$, and e' the corresponding event in $U \cup G$. The algorithm will then backtrack, and start exploring the second configuration. It could then very well reach a configuration that enables e. The updated algorithm will have to re-decide whether e is a cutoff. If it decides that it is not, e.g., because the corresponding event e' has been discarded from $U \cup G$, it could add e to C, and so the second maximal configuration C_2 explored in this way will contain some event enabled by C_1. This implies that C_1 is not a maximal configuration of the prefix $\mathcal{P}'$.

This means essentially that proving that $\mathcal{P}'$ is a *complete prefix* [5] is not a valid strategy for proving Theorem 12, since potentially there exists configurations C of $\mathcal{P}'$ such that $C \notin C_i$ for any $1 \leq i \leq n$.

Alternatively, we could try to reason using a variant of McMillan's standard argument [14, 5, 3] (largely used in the literature about unfoldings for proving that some unfolding prefix is complete). Given a state $s \in \mathit{reach}(M)$, we want to show that there is some configuration C such that

$$\mathit{state}(C) = s \text{ and } C \subseteq C_i \text{ for some } 1 \leq i \leq n. \tag{23}$$

We know that $\mathcal{U}$ contains some configuration C' such that $\mathit{state}(C') = s$. If C' satisfies (23) we are done. If not, the usual argument now finds that C' has a cutoff event, but this does not work in our context: we can easily show that some maximal configuration of $\mathcal{P}'$ enables some event in C' but not in $\mathcal{P}'$ (the wished cutoff), but there is no guarantee that that maximal configuration is one of the C_i's above, so there is no guarantee that the updated algorithm has explicitly declared that event as cutoff.

As a result, we resort to a completely different argument. The main idea is simple. We divide the set of events in $\mathcal{P}'$ in two parts, the *red* events and the *blue* events. Red events are such that the updated algorithm never declares them cutoff, blue events have at least been declared once cutoff and once non-cutoff. We next show two things. First, that the red events contain one representative configuration for every reachable marking (contain a complete prefix). Second, that every configuration formed by red events has been explored by the updated algorithm. Together, these implies Theorem 12.

We start with two definitions.

- Let the *red prefix* be the unique prefix $\mathcal{P}_1 := \langle E_1, <, \# \rangle$ of $\mathcal{U}$ formed by those events e added at least once to U by the updated algorithm and such that every time `Extend'` evaluated the predicate $\mathsf{cutoff}(e, U, G)$, the result was *false*.
- Let the *blue prefix* be the unique prefix $\mathcal{P}_2 := \langle E_2, <, \# \rangle$ of $\mathcal{U}$ such that $E := \bigcup_{1 \leq i \leq n} C_i$.

Observe that $\mathcal{P}_2$ is in fact what we called $\mathcal{P}'$ so far. Notice also that $E_1 \subseteq E_2$.

In § 5.2 we defined the $\mathsf{cutoff}(\cdot)$ predicate using McMillan's size order. Here we redefine it to use an arbitrary *adequate order*. This allows us to prove a more general version of Theorem 12. Let $\prec$ be an adequate order (we skip the definition, the interested reader can find it in [5]) on the configurations of $\mathcal{U}$. We define $\mathsf{cutoff}(e, U, G)$ to hold iff there exists some event $e' \in U \cup G$ such that

$$\mathit{state}([e]) = \mathit{state}([e']) \quad \text{and} \quad [e'] \prec [e]. \tag{24}$$

The *size order* from McMillan, which we used in § 5.2 is indeed adequate [5].

We now need to define the canonical prefix associated with $\prec$ (we refer the reader to [4], to avoid increasing the limited space in the References section, although a better reference would be [Khomenko, Koutny, Vogler 2002]). We give a simplified definition. Given a event $e \in E$, we call it $\prec$-*cutoff* iff there exists some other event $e' \in E$ such that (24) holds. Observe that we now search e' in E and not in $U \cup G$. The $\prec$-*prefix* is the unique $\trianglelefteq$-maximal unfolding prefix that contains no $\prec$-cutoff. It is well known [4] that, (1) the $\prec$-prefix exists and is unique, (2) it is *marking-complete*, i.e., for every $s \in \mathit{reach}(M)$, there is some configuration C in $\prec$-cutoff such that $\mathit{state}(C) = s$.

The key observation now is that all events in $\prec$-prefix are red, i.e., the $\prec$-prefix is a prefix of $\mathcal{P}_1$. Clearly, regardless of the actual contents of U and G when $\mathsf{cutoff}(e, U, G)$ is evaluated, the result will always be *false* if e is not $\prec$-cutoff.

So, in order to prove Theorem 12, it suffices to show that every *red* configuration from $\mathcal{P}_1$ is contained in some node explored the algorithm. We achieve this with Lemma 31 and Lemma 32.

▸ **Lemma 31.** *Let $b \coloneqq \langle C, D, A, e\rangle \in B$ be a node in the call graph and $\hat{C} \subseteq E_1$ an arbitrary* red *configuration in $\mathcal{P}_1$, such that the following two conditions are verified:*

1. *$C \cup \hat{C}$ is a configuration, and*

2. *for any $\tilde{e} \in D$ there is some $e' \in \hat{C}$ such that $\tilde{e} \#^i e'$.*

Then exactly one of the following statements hold:

- *Either b is a leaf node in B, or*
- *for any $\hat{e} \in \hat{C}$ we have $\neg(e \#^i \hat{e})$ and b has a left child, or*
- *for some $\hat{e} \in \hat{C}$ we have $e \#^i e$ and b has a right child.*

Proof. The statement of this lemma is very similar to the one of Lemma 27, the main lemma behind the proof of Theorem 11 (completeness). Consequently the proof is also similar. The proof is by induction on b using the same total order $\lessdot \in B \times B$ that we employed for Lemma 27.

Base case. Node b is the least element in B w.r.t. $\lessdot$. It is therefore the leftmost leaf of the call tree. Then the first item holds.

Step case. Assume that the result holds for any node $\tilde{b} \lessdot b$. If C is maximal, we are done. So assume that C is not maximal. Then b has at least one left child. If we can find some $\hat{e} \in \hat{C}$ such that $\hat{e} \#^i e$, then the second item holds and we are done.

So assume that that for some $\hat{e} \in \hat{C}$ it holds that $\hat{e} \#^i e$. We show that the third item holds in this case. For that we need to show that b has a right child. The rest of this proof accomplishes that, it shows that there is some alternative $J \in \texttt{Alt}(C, D \cup \{e\})$ whenever the algorithm asks for the existence of one.

We define the set

$$F \coloneqq \{e_1, \ldots, e_n\} \coloneqq D \cup \{e\}.$$

This set contains the events that the alternative J needs to *justify*. Let e_i be any event in F. By hypothesis there exists some $e'_i \in \hat{C}$ such that $e_i \#^i e'_i$. Thus, there exists at least one set

$$J \coloneqq [\{e'_1, \ldots, e'_n\}]$$

where $e'_i \in \hat{C}$ and $e_i \#^i e'_i$ for $i \in \{1, \ldots, n\}$. Clearly, $J \subseteq \hat{C}$ and so it is a red configuration of $\mathcal{P}_1$. We remark that J is not uniquely defined, there may be several e'_i to choose for each e_i. For now, take any suitable e'_i without further regard. We will later refine this choice if necessary.

We show now that $J \in \texttt{Alt}(C, D \cup \{e\})$ when function `Alt(·)` is called just before line 11 during the execution of `Explore`(C, D, A). Let $\hat{U}$ be the set of events contained in the variable U exactly when `Alt(·)` is called.

By construction $J \cup C$ is configuration, and contains an event in conflict with any event in $D \cup \{e\}$. We only need to check that $J \subseteq \hat{U}$, i.e., that all events in J were are known (in fact, *remembered*) when function `Alt(·)` is called.

We reason about the call stack when the algorithm is situated at $b = \langle C, D, A, e\rangle$. For $i \in \{1, \ldots, n\}$ let $b_i \coloneqq \langle C_i, D_i, A_i, e_i\rangle \in B$ be the node in the call graph associated to event $e_i \in F$. These nodes are all situated in the unique path from b_0 to b. W.l.o.g. assume (after possible reordering of the index i) that

$$b_0 \rhd^* b_1 \rhd^* b_2 \rhd^* \ldots \rhd^* b_n,$$

where $b_n = b$ and $e_n = e$. Since every event e_i is in $D = D_n$, for $i \in \{1, \dots, n-1\}$, we know that the first step in the path that goes from b_i to b_{i+1} is a *right child*. Also, we remark that by construction we have $\{e_1, \dots, e_{i-1}\} = D_i$ for every $i \in \{2, \dots, n\}$.

We need to show that $e'_i \in \hat{U}$, for $i \in \{1, \dots, n\}$. We consider two cases. Consider the set $D_i = \{e_1, \dots, e_{i-1}\}$. Only two things are possible: either there exists some $j \in \{1, \dots, i-1\}$ such that

$$\#(e_j) \cap \hat{C} \subseteq \#(e_i) \cap \hat{C} \tag{25}$$

holds, or for all $j \in \{1, \dots, i-1\}$ the above statement is false.

- *Case 1*: for all $j \in \{1, \dots, i-1\}$ we have that (25) do not hold. This means that for all such j, some event in $\#(e_j) \cap \hat{C}$ is not in $\#(e_i) \cap \hat{C}$. Consider the set

  $$X_1 := \hat{C} \setminus \#(e_i).$$

  It is a red configuration of $\mathcal{P}_1$, which satisfies the following properties:
  - *Fact 1: set $X_1 \cup C_i \cup \{e_i\}$ is a configuration.* Since $X_1 \cup C_i \subseteq \hat{C} \cup C$, clearly $X_1 \cup C_i$ is a configuration. Also, X_1 has no event in conflict with e_i by construction.
  - *Fact 2: for any $\tilde{e} \in D_i$ there is some $e' \in X_1$ such that $\tilde{e} \#^i e'$.* This holds by construction. For any $\tilde{e} \in D_i = \{e_1, \dots, e_{i-1}\}$ we know that some event in $\#(\tilde{e}) \cap \hat{C}$ is not in $\#(e_i) \cap \hat{C}$, so it is necessarily in X_1.

  Consider the left child $b'_i := \langle C_i \cup \{e_i\}, D_i, \cdot, \cdot \rangle$ of b_i. Every node $\hat{b}$ in the subtree rooted at b'_i (i.e., $b'_i \rhd^* \hat{b}$) is such that $\hat{b} \lessdot b_i \lessdot b$. The induction hypothesis thus applies to $\hat{b}$. By the previous facts, Lemma 32 applied to b'_i and X_1 implies that some leaf (maximal) configuration $C' \supseteq X_1$ has been explored in the subtree rooted at b'_i. Since e'_i is a red event (it will never be declared cutoff) and $e'_i \in ex(C')$, event e'_i will be discovered when exploring C', and will be kept in U as long as e_i remains in U. As a result $e'_i \in \hat{U}$, which we wanted to prove.
- *Case 2*: there is some $j \in \{1, \dots, i-1\}$ such that (25) holds. Let m be the minimum such integer. Consider the set X_2 defined as

  $$X_2 := \hat{C} \setminus \#(e_i) \cup \lceil e'_m \rceil$$

  It is clearly a subset of $\hat{C}$, so it is a red configuration of $\mathcal{P}_1$, and it satisfies the following properties:
  - *Fact 3: set $X_2 \cup C_m \cup \{e_m\}$ is a configuration.* Since $X_2 \cup C_m \subseteq \hat{C} \cup C$, clearly $X_2 \cup C_m$ is a configuration. Also, X_2 has no event in conflict with e_m, since all such events are in $\#(e_i)$ and we have removed them. Observe that by adding $\lceil e'_m \rceil$ we do no add any conflict, as there is no conflict between e_m and any event of $\lceil e'_m \rceil$.
  - *Fact 4: for any $\tilde{e} \in D_m$ there is some $e' \in X_2$ such that $\tilde{e} \#^i e'$.* This holds by construction, as a result of the minimality of m. For any $\tilde{e} \in D_m = \{e_1, \dots, e_{i-m}\}$ we know that (25) do not hold for $\tilde{e}$. So some event in $\#(\tilde{e}) \cap \hat{C}$ is not in $\#(e_i) \cap \hat{C}$, and so it is necessarily in X_2.

  Like before, consider now the left child $b'_m := \langle C_m \cup \{e_m\}, D_m, \cdot, \cdot \rangle$ of b_m. The induction hypothesis applies to any node $\hat{b} \in B$ in the subtree rooted at b'_m (i.e., $b'_m \rhd^* \hat{b}$). By the previous facts, Lemma 32 applied to b'_m and X_2 implies that some leaf (maximal) configuration $C' \supseteq X_2$ has been explored in the subtree rooted at b'_m. Since e'_m is a red event (it will never be declared cutoff) and $e'_m \in ex(C')$, event e'_m will be discovered when exploring C', and will be kept in U as long as e_m remains in U. As a result $e'_m \in \hat{U}$.

We actually wanted to prove that e'_i is in $\hat{U}$. This is now easy. Since $e'_m \in \#(e_i)$, by (25), there is some $\tilde{e} \in \#^i(e_i)$ such that $\tilde{e} \leq e'_m$. Since $\hat{U}$ is causally closed, we have that $\tilde{e} \in \hat{U}$. We have found some event $\tilde{e} \in \hat{U}$ such that $e_i \#^i \tilde{e}$. If $\tilde{e} \neq e'_i$, then we substitute e'_i in J by $\tilde{e}$. This means that in the definition of J we cannot chose any arbitrary e'_i from $\hat{C}$ (as we said before, to keep things simple). But we can always find at least one event in $\hat{C}$ that is in immediate conflict with e_i and is also present in $\hat{U}$. Observe that the choice made for e_i, with $i \in \{1, \ldots, n\}$ has no consequence for the choices made for $j \in \{1, \ldots, i-1\}$. This means that we can always make a choice for index i after having made choices for every $j < i$.

This completes the argument showing that every e'_i (possibly modifying the original choice) is in $\hat{U}$, and shows that $J \subseteq \hat{U}$. This implies, by construction of J, that $J \in \texttt{Alt}(C, D \cup \{e\})$ when the set of events U present in memory equals $\hat{U}$. As a result, the algorithm will do a *right recursive call* and b will have a right child. This is what we wanted to prove. ◂

▸ **Lemma 32.** *Let $b := \langle C, D, \cdot, e\rangle \in B$ be any node the call graph. Let $\hat{C} \subseteq E_1$ be any configuration of $\mathcal{P}_1$, i.e., consisting* only *of red events. Assume that*

- *$C \cup \hat{C}$ is a configuration;*
- *for any $\tilde{e} \in D$ there is some $e' \in \hat{C}$ such that $\tilde{e} \#^i e'$;*
- *Lemma 31 holds on every node in the subtree rooted at b.*

Then there exist in B a node $b' := \langle C', \cdot, \cdot, \cdot\rangle$ such that $b \rhd^* b'$ and $\hat{C} \subseteq C'$.

Proof. Assume that Lemma 31 holds on any node $b'' \in B$ such that $b \rhd^* b''$, i.e., all nodes in the subtree rooted at b. By hypothesis we can apply Lemma 31 to b and $\hat{C}$. If C is maximal, i.e., the algorithm do not find any non-cutoff extension of C, then we have that $\hat{C} \subseteq C$, as otherwise any event in $\hat{C} \setminus C$ would be non-cutoff (as it is red) and would be enabled at C (because $\hat{C} \cup C$ is a configuration). So if b is a leaf, then we can take $b' := b$.

If not, then e is enabled at C and there is at least a left child. Two things can happen now. Either e is in conflict with some event in $\hat{C}$ or not.

If e is not in conflict with any event in $\hat{C}$, then the left child $b_1 := \langle C_1, D_1, \cdot, e_1\rangle$, with $C_1 := C \cup \{e\}$ and $D_1 := D$, is such that $C_1 \cup \hat{C}$ is a configuration, and $\hat{C}$ contains some event in conflict with every event in D_1. Furthermore Lemma 31 applies to b_1 as well.

If e is in conflict with some event in $\hat{C}$, then by Lemma 31 we know that b has a right child $b_1 := \langle C_1, D_1, \cdot, e_1\rangle$, with $C_1 := C$ and $D_1 := D \cup \{e\}$. Like before, $C_1 \cup \hat{C}$ is a configuration and for any event in D_1 we have another one in $\hat{C}$ in conflict with it.

In any case, if C_1 is maximal, then it holds that $\hat{C} \subseteq C_1$ and we are done. If not, we can reapply Lemma 31 at b_1 and make one more step into one of the children b_2 of b_1. If C_2 still do not contain $\hat{C}$, then we need to repeat the argument starting from b_2 only a finite number n of times until we reach a node $b_n := \langle C_n, D_n, \cdot, \cdot\rangle$ where b_n has no further children in the call tree (i.e., $en(C_n)$ is either empty or contains only cutoff events). This is because every time we repeat the argument on a non-leaf node b_i we advance one step down in the call tree, and all paths in the tree are finite. So eventually we find a leaf node b_n, which, as argued earlier, satisfies that $\hat{C} \subseteq C_n$, and we can take $b' := b_n$. ◂

▸ **Theorem 12** (Completeness). *For any reachable state $s \in reach(M)$, Alg. 1 updated with the cutoff mechanism described above explores one configuration C such that for some $C' \subseteq C$ it holds that $state(C') = s$.*

Proof. Let $\mathcal{P}$ be an unfolding prefix constructed with the classic saturation-based unfolding algorithm, using the standard cutoff strategy in combination with an arbitrary adequate

order $\prec$: an event e is a *classic-cutoff* if there is another event e' in $\mathcal{U}_M$ such that $\mathit{state}([e]) = \mathit{state}([e'])$ and $[e'] \prec [e]$. By construction all events in $\mathcal{P}$ are red, so they are in $\mathcal{P}_1$.

Let $\langle B, \triangleright \rangle$ be the call tree associated with one execution of Alg. 1 retrofitted with the cutoff mechanism. Let $s \in \mathit{reach}(M)$ be an arbitrary state of the system. Owing to the properties of $\mathcal{P}_M$ [5], there is a configuration $\hat{C}$ in $\mathcal{P}$ such that $\mathit{state}(\hat{C}) = s$. Such a configuration is in $\mathcal{P}_1$.

Now, Lemma 32 applies to the initial node $b_0 \in B$ and $\hat{C}$, and guarantees that the algorithm will visit a node $b := \langle C, \cdot, \cdot, \cdot \rangle \in B$ such that such that $\hat{C} \subseteq C$. This is what we wanted to prove. ◂