Unfolding-based Partial Order Reduction

César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening

1 Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, France
2 Department of Computer Science, University of Oxford, UK

Abstract. TBD

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, a scenario often referred as the state-space explosion problem. Existing approaches to address this problem can essentially be classified as either partial-order reduction (POR) techniques or unfolding-based methods.

POR techniques [Pel93, FG05] attempt to construct a fragment of the state space where irrelevant executions of the original system are discarded. Such executions are identified using an independence relation between transitions of the system. For instance, consider the state-space shown in Fig. 1a. Since transitions $a$ and $b$ reach the same state when executed from $s_0$, their execution can be performed in any order and will provably reach the same final state. This means that the execution $a.c.b$ does not need to be in the reduced fragment of the state space as long as $a.b.c$ is present. Figure 1b illustrates a reduced state space that POR could explore, where at least one sequence of commutative transitions is represented. In this sense, POR methods have also been called model checking using representatives [Pel93], and some authors argue that they should actually be considered a technique for exploring representative executions rather than a method derived from partial-order semantics [Val98].

An alternative approach to POR emerges from the use of partial-order semantics [NPW81], where the state space is represented by a partial order instead of a graph that interleaves concurrent actions. In the former, concurrent actions are simply left unordered. Petri net unfoldings are arguably the most prominent verification technique that has arisen from this idea [EH08], and Ken McMillan was the first to demonstrate its utility in verification [McM]. The central principles of unfoldings can be viewed as follows. First, one identifies a collection of partial orders whose linearizations cover and partition the set of all interleaved executions of the system. These partial orders, called processes [Pet77] in Petri net theory or dependence graphs [Maz87] in trace theory, are then bundled together into the so-called unfolding semantics, a partially ordered structure [NPW81] equipped with a conflict relation that allows to retrieve from the unfolding each
Fig. 1: Alternative representations of the state-space of a concurrent system.

constituent partial order. Referring back to our example, Fig. 1c shows a possible collection of partial orders that describe the behaviour of Fig. 1a. The unfolding, shown in Fig. 1d, merges (the longest common prefixes of) all of them. The conflict relation between events is shown with dotted edges.

POR methods and net unfoldings not only address similar problems, but exploit fundamentally the same opportunities for reduction available in the state space, the commutativity of transitions. Both fields have co-existed for over two decades, yet, to the best of our knowledge, a systematic study aiming at establishing a formal connection between them is missing.

The first step in this direction is to formulate both methods within the same framework. PORs are often presented for abstract computation models and remain decoupled of the specific formalism to which they will be applied. However, unfoldings have almost exclusively been considered for Petri nets, where the unfolding construction is highly entangled with the syntax of the net. [TODO: Refine this statement; they might have been used, but they have never been formalized on a general model of computation, all definitions were entangled with specific details of the specific model of computation on which they were formulated]

We perform the comparison on an abstract model of computation in order to make the connection between both methods widely applicable. As a result, we need to define unfolding semantics using exactly the same semantic information that POR methods employ: the commutativity of transitions.

Furthermore, we believe that one important fact explaining why unfoldings have almost exclusively remained in the context of Petri nets is the lack of a general definition for broader classes of systems (with the notable exception of [LB99], where unfolding semantics are given to a process algebra).

Therefore, this paper makes the following key contributions:

– We develop a general model of computation (§ 2) and identify under which assumptions both techniques compute comparable representations of concurrent behaviour (§ 3).
– We define unfolding semantics parameterized by an arbitrary independence relation and show a direct application to shared-variable programs (??).
We compare the representations that both unfoldings and POR methods construct with respect to their size and cost of information retrieval. Based on our findings, we address the question of whether POR should or not be actually considered model checking with representatives and discuss a number of open problems relevant to establishing a deeper understanding of the precise connection between POR and unfolding techniques.

2 Model of Computation

We consider an abstract model of concurrent computation, general enough to capture any reasonable means of process interaction, such as shared memory or message passing. Our model represents a finite number of concurrent, finite-state processes that communicate by means of some finite shared state.

Formally, a system is a tuple $M := (\Sigma, T, \tilde{s})$ formed by a set $\Sigma$ of global states, a set $T$ of transitions, and some initial global state $\tilde{s} \in \Sigma$. Each transition $t : \Sigma \rightarrow \Sigma$ 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 $enabl(s)$ denote the set of transitions enabled at $s$. The interleaving semantics of $M$ is the edge-labelled directed graph $S_M := (\Sigma, \rightarrow, \tilde{s})$ where $\Sigma$ are the global states, $\tilde{s}$ is the initial state and $\rightarrow \subseteq \Sigma \times T \times \Sigma$ contains a triple $(s, t, s')$, denoted by $s \xrightarrow{t} s'$, iff $t$ is enabled at $s$ and $s' = t(s)$. Given two states $s, s' \in \Sigma$, and $\sigma := t_1.t_2 \ldots t_n \in T^*$ ($t_1$ concatenated with $t_2, \ldots$ until $t_n$), we denote by $s \xrightarrow{\sigma} s'$ the fact that there exist states $s_1, \ldots, s_{n-1} \in \Sigma$ such that $s \xrightarrow{t_1} s_1, \ldots, s_{n-1} \xrightarrow{t_n} s'$.

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 $state(\sigma)$ the state $s$ that $\sigma$ reaches, and by $runs(M)$ the set of runs of $M$, also referred to as the interleaving space. A state $s \in \Sigma$ is reachable if $s = state(\sigma)$ for some $\sigma \in runs(M)$; it is a deadlock if $enabl(s) = \emptyset$, and in that case $\sigma$ is called maximal, or deadlocking. We let $reach(M)$ denote the set of reachable states in $M$.

For the rest of the paper, we fix a system $M := (\Sigma, T, \tilde{s})$ and assume that (i) $reach(M)$ is finite and that (ii) each transition in $T$ is a function decidable in polynomial time.

3 Partial Order Reductions

Assume that we are concerned only with the reachability of deadlock states in a program. If two transitions of the program commute at one state, such as $a$ and $b$ at state $s_3$ in Fig. 1a, it seems sufficient to explore no more than one of their interleavings. While commutativity alone may not not always be sufficient, it remains a central idea behind most POR methods.

More formally, given two transitions $t, t' \in T$ and one state $s \in \Sigma$, we say that $t, t'$ commute at $s$ exactly when
if \( t \in \text{enabl}(s) \) and \( s \xrightarrow{t} s' \), then \( t' \in \text{enabl}(s) \) iff \( t' \in \text{enabl}(s') \); and

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

For instance, \( b \) and \( c \) commute at state \( s_1 \) but not at \( s_0 \). Also, \( a \) and \( b \) vacuously commute at \( s_5 \). Commutativity of transitions at states identifies an equivalence relation on the set \( \text{runs}(M) \). Two runs \( \sigma \) and \( \sigma' \) of the same length are equivalent, written \( \sigma \equiv \sigma' \), if they can be obtained by swapping transitions that commute.

To explore any deadlock state \( d \) such that \( \text{state}(\sigma) = d \), it is then sufficient to explore an arbitrary run \( \sigma' \) such that \( \sigma' \equiv \sigma \), as clearly both \( \sigma \) and \( \sigma' \) reach \( d \).

POR methods constitute a large family of algorithms exploring a subgraph of \( S_M \) provably large enough to contain all local states. This is achieved by means of a selective search, at any any reached state \( s \), the algorithm selects a (provably large enough) subset \( P \subseteq \text{enabl}(s) \), and explores recursively every state \( s' \) with \( t \in P \) and \( s \xrightarrow{t} s' \). Deciding whether \( t \) should be taken in \( P \) is in general a difficult problem [God96]. As a simplification, all POR methods are parameterized by an independence relation that soundly under-approximates the commutativity of transitions: if \( t \) and \( t' \) are declared independent at \( s \), then they must commute at \( s \), but not necessarily the other way around. When independence relation does not require at which states the transitions \( t \) and \( t' \) are independent, then they commute at all reachable states. Such a (binary) relation is called unconditional independence, as opposed to the (ternary) conditional independence that declares transitions independent at particular states.

Formally, an unconditional independence relation on \( M \) is any symmetric and irreflexive relation \( \Diamond \subseteq T \times T \) satisfying that if \( t \Diamond t' \), then \( t \) and \( t' \) commute at every reachable state \( s \in \Sigma \). If \( t, t' \) are not independent according to \( \Diamond \), then they are dependent, denoted by \( t \triangleleft t' \).

Unconditional independence identifies an equivalence relation \( \equiv \) on the set \( \text{runs}(M) \). Formally, \( \equiv \) is defined as the transitive closure of the relation \( \equiv^1 \), which in turn is defined as \( \equiv^1 \equiv^1 \equiv \equiv \) 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 \( \equiv \) refines \( \equiv \), i.e., if \( \sigma \equiv \sigma' \), then \( \sigma \equiv \sigma' \).

Given any run \( \sigma \in \text{runs}(M) \), the set of runs equivalent to \( \sigma \), i.e., the equivalence class of \( \equiv \) to which \( \sigma \) belongs, is called a Mazurkiewicz trace [Maz87], and denoted by \( T_{\sigma} \). It is well known that each Mazurkiewicz trace can equivalently be seen as a labelled partial order \( D_{\sigma} \) traditionally called the dependence graph (see [Maz87] for a formalization) satisfying that a run belongs to the trace if and only if it is a linearization of \( D_{\sigma} \).

Sleep sets are another technique for state-space reduction. However, unlike selective exploration from a state, they rely on past explorations to dictate what must be explored from a certain state. For instance, assume that at state \( s \) two transitions \( t, t' \) are enabled such that \( s \xrightarrow{t} s' \) and \( s \xrightarrow{t'} s'' \). While backtracking from \( s' \) in a depth-first exploration, the technique infers that from \( s \) we have already explored \( t \). Hence, \( t \) must be disallowed to fire from all successors of \( s \) (except \( s' \)) until a transition dependent on \( t \) was fired from a state reachable from \( s \). This condition imposes that independent actions are not interleaved in all
possible orders. Sleep sets are orthogonal to selective search and can be soundly combined to further reduce the state space. In [God96], a depth-first search (DFS) algorithm that uses both selective search and sleep sets is presented.

4 Parametric Partial Order Semantics

In this section, we define unfolding semantics for our model of computation. The semantics will be parameterized by an independence relation. Unfoldings are, conceptually, a tree-like structure of partial orders. The main intuition behind parameterizing the semantics with an independence relation $\diamond$ is that each constituent partial order will correspond to one dependence graph $D_{\diamond, \sigma}$ for some $\sigma \in \text{runs}(M)$.

For the rest of this section, let $\diamond$ be an arbitrary unconditional independence relation on $M$. We use prime event structures [NPW81], a non-sequential, event-based model of concurrency, to define the unfolding of $M$.

**Definition 1 (LES).** Given a set $A$, an $A$-labelled event structure ($A$-LES, or LES in short) is a tuple $\mathcal{E} := (E, <, \#, h)$ where $E$ is a set of events, $< \subseteq E \times E$ is a strict partial order on $E$, called causality relation, $h : E \rightarrow A$ labels every event with an event, and $\# \subseteq E \times E$ is the symmetric, irreflexive conflict relation, satisfying

1. for all $e \in E$, $\{ e' \in E : e' < e \}$ is finite, and
2. for all $e, e', e'' \in E$, if $e \neq e'$ and $e' < e''$, then $e \neq e''$.

The *causes* of an event $e \in E$ is the set $[e] := \{ e' \in E : e' < e \}$ 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:

1. for all $e \in C$ we have $[e] \subseteq C$ (causally closed)
2. for all $e, e' \in C$, it holds that $\neg e \neq e'$ (conflict free).

In particular, the *local configuration* of $e$ is the $\subseteq$-minimal configuration that contains $e$, i.e. $[e] := [e] \cup \{ e \}$. Observe that $[e]$ and $[e]$ are indeed configurations, as $\#$ is irreflexive. We denote the maximal causal events of a configuration $C$ as $\text{max}(C)$. Two events $e, e'$ are in *immediate conflict*, $e \neq e'$ and $[e] \cup [e']$ is conflict-free. For the rest of this work we will only consider $A$-labelled LESs where the set $A := T \cup \{ \varepsilon \}$ of labels is the set $T$ of transitions of $M$ together with some special *empty string* $\varepsilon$.

Given two LESs $\mathcal{E} := (E, <, \#, h)$ and $\mathcal{E}' := (E', <', \#', h')$, we say that $\mathcal{E}$ is a *prefix* of $\mathcal{E}'$, written $\mathcal{E} \preceq \mathcal{E}'$, when $E \subseteq E'$, $< = <' \cap (E \times E)$, $\# = \#' \cap (E \times E)$, $h = h' \cap (E \times E)$, and $E$ is causally closed in $\mathcal{E}'$, i.e., $E$ satisfies that for any $e \in E$ and any $e' \in E'$ with $e' < e$ we have $e' \in E$.

The semantics we propose unrolls the system $M$ into a LES whose events are labelled by transitions of $M$. This LES, formally defined below, will be called the unfolding of $M$, and its configurations will be in one-to-one correspondence with the collection of dependence graphs $D_{\diamond, \sigma}$ for $\sigma \in \text{runs}(M)$. 
Since every configuration $C$ of the unfolding corresponds to a Mazurkiewicz trace, it represents a set of executions. For an LES $\langle E, \prec, \#, h \rangle$, we define the set of *interleavings* of $C$ as $\text{inter}(C) := \{h(e_1), \ldots, h(e_n) : e_i \prec e_j \implies i < j\}$. Notice that for arbitrary LES $\text{inter}(C)$ may contain sequences that are not executions of $M$. However, the definition of the unfolding will ensure that $\text{inter}(C) \in \text{runs}(M)$. Additionally, since all sequences in $\text{inter}(C)$ belong to the same trace, all of them reach the same state. Abusing the notation, we define $\text{state}(\sigma)$ if $\sigma \in \text{inter}(C)$. Observe that the definition is neither well-given nor unique for arbitrary LES, but will be so for the unfolding as we will show.

We start now defining the unfolding of $M$. Each event will be uniquely identified by a canonical name of the form $e := (t, H)$, where $t \in T$ is a transition of $M$ and $H$ a configuration of the unfolding. Intuitively, $e$ represents the occurrence of $t$ after the history $H$. The history will coincide with the causes $[e]$ of $e$.

The definition will be inductive. The base case inserts into the unfolding a special *bottom event* $\bot$ on which every event causally depends. The inductive case iteratively extends the unfolding one event at a time. At every step a transition $t$ is added with some history $H$ to the *version* of the unfolding constructed on the previous step. The set $\mathcal{H}_{E,t}$ of candidate histories for a transition $t$ in an LES $E$ will contain all configurations $H$ of $E$ such that

- the transition $t$ is enabled at $\text{state}(H)$, and
- either $H = \{\bot\}$ or for all events $e \in \max(H)$, we have that $h(e) \nleq t$,

where $h$ is the labelling function in $E$. 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 a same configuration, as configurations represent equivalent executions. We will solve this by introducing a conflict between $e$ and $e'$. The set $\mathcal{K}_{E,e}$ of *events conflicting* with $e := (t, H)$ will thus contain any event $e'$ in $E$ with $e' \notin [e]$ and $-e \prec e'$ and $t \nleq h(e')$.

Let $F := \{(E_1, <, 1, h_1), (E_2, <, 2, h_2), \ldots\}$ be a set of A-LESs. We define the *union* of all LES in $F$ as the LES

$$\text{union}(F) := \{(E_1 \cup E_2 \cup \ldots), (<_1 \cup<_2 \cup \ldots), (#_1 \cup #_2 \cup \ldots), (h_1 \cup h_2 \cup \ldots)\}$$

where we see the labelling function of a LES as a set. Since every element of $F$ is a A-LES, clearly $\text{union}(F)$ is also a A-LES. (1) and (2) are trivially satisfied.

Observe that for unfolding prefixes, whose events are pairs of the form $(t, H)$, the union of two or more unfolding prefixes will merge *equal events*. Indeed, two events $e_1 := (t_1, H_1)$ and $e_2 := (t_2, H_2)$ are equal iff $t_1 = t_2$ and $H_1 = H_2$.

Following common practice [Eng91], the definition proceeds in two steps. We first define (Definition 2) the collection of all LES that are prefixes of the unfolding. Then we show that there exists only one $\sim$-maximal element in the collection, and define it to be the unfolding (Definition 3).

**Definition 2 (Unfolding prefixes).** The set of unfolding prefixes of $M$ is the smallest set of LESs that satisfies the following conditions:
1. The LES that has exactly one event \( \perp \), empty causality and conflict relation, and \( h(\perp) := \varepsilon \), is an unfolding prefix.

2. Assume that \( \mathcal{E} \) is an unfolding prefix containing a history \( H \in \mathcal{H}_{\mathcal{E},t} \) for some transition \( t \in T \). Then, the LES \( (E,<,\#,h) \) resulting from extending \( \mathcal{E} \) with a new event \( e := (t,H) \) and satisfying the following constraints is also an unfolding prefix of \( M \):
   - for all \( e' \in H \), we have \( e' < e \);
   - for all \( e' \in \mathcal{K}_{\mathcal{E},e} \), we have \( e \neq e' \), and
   - \( h(e) := t \).

3. For any (finite or infinite) set \( X \) of unfolding prefixes, the LES union \( (X) \) is also an unfolding prefix.

Each unfolding prefix intuitively represents dependence graphs associated to executions of \( M \) of bounded lengths. One can view the construction of the unfolding from the event \( \perp \) at the start of the execution (condition 1) and then iteratively applying condition 2 for any transition until no more events can be added to the current unfolding prefix. Observe that the set of unfolding prefixes is finite because we have assumed that the interleaving semantics is acyclic.

Our first task is verifying that each unfolding prefix is indeed a LES:

**Lemma 1.** [César: Marcelo, start writing a proof for this lemma, should be simple] For any unfolding prefix \( (E,<,\#,h) \) we have the following:

1. The relation \(<\) is a strict partial order.
2. The relation \(\#\) is irreflexive.

As stated earlier, the configurations of every unfolding prefix \( \mathcal{P} \) of \( M \) need to correspond the Mazurkiewicz traces of the system. In particular, we need to verify that for any configuration \( C \) of \( \mathcal{P} \), any interleaving in \( \text{inter}(C) \) is a run of \( M \) and that any two interleavings in \( \text{inter}(C) \) reach the same state.

**Lemma 2.** [César: Marcelo, then prove this one. Observe that \( C \) is now a possibly infinite configuration, the proof I wrote was for \( C \) being finite.] For any configuration \( C \) of an unfolding prefix, \( \text{inter}(C) \subseteq \text{runs}(M) \). Furthermore, if \( C \) is finite, then for any \( \sigma_1, \sigma_2 \in \text{inter}(C) \), we have \( \text{state}(\sigma_1) = \text{state}(\sigma_2) \).

This shows that the definition of \( \text{inter}(C) \) and \( \text{state}(C) \) is well-given when \( C \) belongs to an unfolding prefix.

Our second task is defining the unfolding of the system. The set of unfolding prefixes is ordered by the prefix relation \( \leq \). We first show that any two unfolding prefixes have a join prefix containing all events of both.

**Lemma 3.** [César: Then prove this one] The set of unfolding prefixes of \( M \) equipped with the prefix relation \( \leq \) is a complete lattice.

Lemma 3 essentially implies that there exist a unique and \( \leq \)-maximal unfolding prefix:
Definition 3 (Unfolding). The unfolding $U_M$ of $M$ is the unique $\leq$-maximal element in the set of unfolding prefixes of $M$.

Theorem 1. [César: This is a breakfast-proof corollary of Lemma 3] The unfolding $U_M$ exists and is unique.

A second important result is that the unfolding is complete, i.e., every run of the system is uniquely represented by a configuration of the unfolding.

Theorem 2. [César: Finally go for this one, see my comment at the beginning of the current proof] For any non-empty run $\sigma$ of $M$, there exists a unique configuration $C$ of $U_M$ such that $\sigma \in \text{inter}(C)$.

4.1 Examples

We now illustrate the properties of the unfolding structure defined in the previous section. Considering the example in Fig. 1, intuitively one can view the unfolding in Fig. 1d as a compact representation where all dependence graphs in Fig. 1c have been merged. Each partial order represents the Mazurkiewicz trace of a deadlocking execution, and each corresponds to a $\subseteq$-maximal configuration in the unfolding. Every interleaving of a configuration reaches the same global state. The structure of each partial order is preserved in the unfolding, and conflicts are introduced between events of different partial orders that have not been merged in order to prevent that a configuration does not correspond to a partial order.

Since our unfolding definition is parameterized by an independence relation, we can devise a possible instantiation of the unfolding semantics to, e.g., programs with shared variables. In this model, one often considers independent two operations over different variables as well as and read operations over the same variable. Hence, there are write-write and read-write dependencies over the same shared variable. With this independence relation and the concurrent program in Fig. 2, our definition constructs the unfolding shown in Fig. 3. The system is composed of four processes $p, q, r, s$ and three shared variables $x, y$ and $z$. We represent the $i$-th transition of a process $u$ as $u_i$. In this system, there are three data races: a write-write dependence $p_1 \diamond s_2$, and two write-read dependencies: $q_1 \diamond r_1$ and $r_2 \diamond s_1$. For example, $p_1 \diamond q_1$, since they write to different variables.

Initially: $x := y := z := 0$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$:</td>
<td>$q$:</td>
<td>$r$:</td>
<td>$s$:</td>
</tr>
<tr>
<td>$x := 1; $</td>
<td>$y := 1; $</td>
<td>if($y=0$) then</td>
<td>if($z=1$) then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$z := 1;$</td>
<td>$x := 2;$</td>
</tr>
</tbody>
</table>

Fig. 2: Shared-variable concurrent system
Fig. 3: Unfolding of example in Fig. 2.

Since \( p_1 \diamond q_1 \), \( e_1 \) and \( e_2 \) are concurrent in the unfolding. Consider the unfolding prefix \( P := \langle \{1, e_2\}, \{1, e_2\}, \emptyset, h \rangle \) that represents the state where only \( q \) has been executed. At this step, \( H_P, r_1 := \{1\}, \{1, e_2\} \). Therefore, the second condition to extend \( P \) is satisfied for \( t \equal{} r_1 \) and \( H = \{1\} \), which will introduce a new event \( \hat{e}_3 := \{r_1, \{1\}\} \). The application of the rule will extend \( P \) to \( P' := \langle \{1, e_2, \hat{e}_3\}, \{1, e_2\}, \{1, \hat{e}_3\}, \{e_2, \hat{e}_3\} \rangle, h' \rangle \). Observe that \( D_{P, \hat{e}_3} := \{e_2\} \) because \( r_1 \diamond q_1 \). The event \( e_3 \) represents the other possible history for transition \( r_1 \). For simplicity, we do not represent conflicts between events of the same transition in Fig. 3, although they will be contained in \#. The unfolding of Fig. 3 contains four maximal configurations \( \{e_1, e_2, e_3, e_4\}, \{e_1, \hat{e}_3, \hat{e}_2, e_4, e_5\}, \{e_1, \hat{e}_3, \hat{e}_2, e_5, \hat{e}_4, e_6\}, \{\hat{e}_3, e_2, e_5, \hat{e}_4, e_6, \hat{e}_1\} \), which respectively correspond to the Mazurkiewicz traces that reach the set of states \( \{x := 1; y := 1; z := 0\}, \{x := 1; y := 1; z := 1\}, \{x := 1; y := 1; z := 1\}, \{x := 1; y := 1; z := 1\} \).

### 4.2 Comparison to Petri Net Unfoldings

The semantics introduced in previous section is parametrized by an independence relation. However, the classical unfolding semantics for Petri nets [ERV02] (or synchronizations of state machines [EH08], or process algebras [LB99]) employs a fixed independence relation, specifically the complement of the following dependence relation. Two transitions of a Petri net are dependent, \( t \diamond_n t' \), iff

\[ (t^* \cap t'^* = \emptyset) \text{ or } (t^* \cap t = \emptyset) \text{ or } (t'^* \cap t = \emptyset), \]

where \( t \) and \( t^* \) are resp. the preset and postset of \( t \). Classic Petri net unfoldings are therefore a specific instantiation of our semantics.

A well known limitation of classic unfoldings is transitions that “read” places, e.g., \( t_1 \) and \( t_2 \) in Fig. 4a. Transitions \( t_1 \) and \( t_2 \) are dependent according to \( \diamond_n \), and so the classic unfolding, Fig. 4b, sequentializes all their occurrences. A solution to this is the so-called place replication (PR) unfolding [MR95], or alternatively contextual unfoldings (which anyway internally are of asymptotically the same size as the PR-unfolding).
An advantage of our parametric semantics is that we can now restrict the
dependence relation so as to produce a parametric unfolding exponentially more
compact than the classic one. In fact, of the same size as the PR-unfolding.
The idea is to define a smaller dependency relation $\diamond_n' \subset \diamond_n$ which declares
independent transitions that “read” common places. Accordingly, we would
have $t_1 \diamond_n' t_2$, and the parametric semantics $U_{M, \diamond_n'}$, shown in Fig. 4c, would
leave $t_1$ and $t_2$ concurrent (as the PR unfolding would do). With $n$ reading trans-
itions, the classic unfolding would have $O(n!)$ events while our unfolding would
have $O(2^n)$. The point here is that our semantics accommodate for a potentially
better notion of independence without resorting to specific ad-hoc tricks.

Furthermore, although our semantics use unconditional independence, we
conjecture that this is in fact not necessary, and an adequately restricted conditional
dependence would suffice, e.g., the one of [KP92]. Gains achieved in such
setting would probably be impossible with classic unfoldings, or variants thereof.

Finally, our parametric semantics help gaining a closer insight as to what
PORs and unfoldings fundamentally do in terms of the other.

5 Stateless Unfolding Exploration Algorithms

Existing unfolding construction algorithms work by growing an unfolding prefix,
all configurations at a time. Extensions are added to any of the existing config-
urations of the prefix. Although this is often efficient, it is also a liability, as it
is impossible to remove an event from the structure once it has been added. In
contrast, PORs work with only one configuration at a time, making use of ade-
quate mechanisms to backtrack and switch to the next configuration whenever a
maximal configuration has been reached. Computing extensions is more efficient
than for unfoldings, but they possibly visit the same events exponentially many
times (unlike unfoldings).

What would be nice :) FIXME is to have an unfolding exploration algorithm
that has the benefits of both: adding events is cheap, doing so do not becomes a
liability, and we can make use of as much memory as it is available. In this section
we present two variants of algorithm that achieves this. It explores exactly once

---

Fig. 4: Classic Petri net unfolding vs. our parametric semantics.
every configuration of the unfolding, works on one configuration at a time and is able to cache every event already visited for future re-exploration.

It is therefore stateless and optimal (in the sense of [AAJS14]).

For the rest of the paper, fix an unconditional independence relation $\Diamond$ for $M$. This determines an unfolding semantics $U_{\Diamond,M} := (E, <, \#, h)$ that we will abbreviate as $U$. For this section we assume that $U$ is finite, i.e., that all computations of $M$ terminate. This is to ease presentation, we relax this assumption in § 6.3.

We set some new definitions. Let $C$ be a configuration of $U$. We let $en(C)$ denote the set of events enabled by $C$, i.e., those corresponding to the transitions enabled at state($C$). Formally, $en(C) := \{ e \in E : e \notin C \land (C \cup \{ e \}) \in conf(U) \}$. The extensions of $C$, written $ex(C)$, are all those events outside $C$ whose causes are included in $C$. Formally, $ex(C) := \{ e \in E : e \notin C \land e \subseteq C \}$. Remark that events enabled by $C$ are always extensions of $C$, i.e., $en(C) \subseteq ex(C)$. All those extensions which are not enabled are conflicting extensions, formally defined as $cex(C) := \{ e \in ex(C) : \exists e' \in C, e \neq ^i e' \}$. Clearly, for every configuration $C$ the sets $en(C)$ and $cex(C)$ partition the set $ex(C)$ in two disjoint parts.

Lastly, we define $\#^i(e) := \{ e' \in E : e \neq ^i e' \}$, 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 $U$.

### 5.1 Lazy Algorithm

Algorithm 1 presents a stateless POR algorithm that recursively constructs all configurations of $U$. The main procedure, $Explore(C, D, A)$, stores in its first parameter $C$ the configuration at which the algorithm sits right now. Set $D$ (for disabled) intuitively summarizes what configurations have already been explored and enables $Explore()$ avoiding repeating work. It plays a role similar to that of a sleep set. Set $A$ (for add) is used at certain occasions to force the exploration to progress in certain direction.

Additionally, the algorithm uses a global set $U$ where it stores all events presently known to the algorithm. Whenever an event can safely be discarded from memory it is moved from $U$ to $G$ (for garbage). Once in $G$, it can be removed from memory at any time, or left there expecting that it will be re-explored (re-
inserted in $U$), thus avoiding work to re-construct it. Set $G$ can alternatively be viewed as a cache memory.

**Algorithm 1:** Lazy POR based on unfolding semantics.

1. Initially, set $U := \{1\}$, set $G := \emptyset$, and call $\text{Explore}(\{1\}, \emptyset, \emptyset)$.

2. **Procedure** $\text{Explore}(C, D, A)$
   
   3. **Procedure** $\text{Extend}(C)$
   
   4. if $\text{en}(C) = \emptyset$ return
   
   5. if $A = \emptyset$
   
   6. | Choose $e$ from $\text{en}(C)$
   
   7. else
   
   8. | Choose $e$ from $A \cap \text{en}(C)$
   
   9. $\text{Explore}(C \cup \{e\}, D, A \setminus \{e\})$
   
   10. if $\exists J \in \text{Alt}(C, D \cup \{e\})$
   
   11. | $\text{Explore}(C, D \cup \{e\}, J \setminus C)$
   
   12. **Remove**($e, C, D$)

3. - it first updates $U$ with all extensions of $C$ - it then chooses an enabled event, possibly from $A$ if there is one to take - then 2 recursive calls, we refer them as the left one and the right one - always the first, including the event - possibly the second, if it finds it needs to - alternatives:

**Definition 4 (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
- for all events $e \in D$, there is some $e' \in C \cup J$ such that $e' \in \#_{U}(e)$; (6)

We denote by $\text{Alt}(C, D)$ the set of all such $J$.

- intuition about alternative - justifies $D$, and forces itself on the right call to explore that justification - $D$ can thus be seen as a request to itself, it asks itself not to explore any configuration including $C$ and $e$, as all have already been explored on the left call - remark that we need $e'$s to decide, we need to store them!

- after coming back from the right call, we have to possibly remove $e$ and/or events in conflict with $e$, according to $\text{Remove()}$ and the following definition

We define the set of events kept by algorithm 1 at any time as

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

- the above helps to clean set $U$ of events that are now unnecessary to continue the exploration, i.e., to compute alternatives in the future! - intuition: if we discard too much here the algorithm would become incomplete, sometimes it wouldn’t go right
- the algorithm is non-deterministic - introduce the theorems below

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.

**Theorem 3 (Termination).** Regardless of its input, algorithm 1 always stops.

**Theorem 4 (Optimality).** Let $\tilde{C}$ be a maximal configuration of $U_M$. Then Explore($\cdot$, $\cdot$, $\cdot$) is called at most once with its first parameter being equal to $\tilde{C}$.

**Theorem 5 (Completeness).** Let $\tilde{C}$ be a maximal configuration of $U_M$. Then Explore($\cdot$, $\cdot$, $\cdot$) is called at least once with its first parameter being equal to $\tilde{C}$.

### 5.2 Eager Algorithm

- Say that $C$ is a stack - Introduce notation $C(e)$ - Define A justifies B with immediate conflict # - We need $k(e)$ still here because of the justifies

**Algorithm 2:** Eager POR based on unfolding semantics.

1. Initially, set $U = G \leftarrow \emptyset$ and call Explore2($\{1\}$, $\emptyset$, $\emptyset$)

2. **Procedure** Explore2($C, D, A$)
   
   3. \hspace{1em} Ext($C$)
   4. \hspace{1em} if $en(C) = \emptyset$
   5. \hspace{2em} \textbf{Save}($C$)
   6. \hspace{2em} return
   7. \hspace{1em} if $A = \emptyset$
   8. \hspace{2em} \hspace{1em} Choose $e$ from $en(C)$
   9. \hspace{2em} else
   10. \hspace{2em} \hspace{2em} Choose $e$ from $A \cap en(C)$
   11. \hspace{2em} \hspace{2em} Set $C(e), D(e), A(e)$ to $C$, $D$, $\emptyset$
   12. \hspace{2em} \hspace{2em} Explore2($C \cup \{e\}, D, A \cup \{e\}$)
   13. \hspace{2em} \hspace{2em} if $\exists J \in \text{Alt}_2(C, D)$
   14. \hspace{2em} \hspace{2em} \hspace{2em} Explore2($C, D \cup \{e\}, J \setminus C$)
   15. \hspace{2em} \hspace{2em} Remove2($e, C, D$)
   16. \hspace{1em} \textbf{Procedure} Save($C$)
   17. \hspace{2em} \hspace{1em} foreach $e \in C$ and $e' \in #_C^{-}(e)$
   18. \hspace{2em} \hspace{2em} Let $J := C \setminus #(e') \cup \{e'\}$
   19. \hspace{2em} \hspace{2em} if $J \in \text{Alt}(C(e), D(e))$
   20. \hspace{2em} \hspace{2em} \hspace{2em} Add $J$ to $A(e)$
   21. \hspace{1em} \textbf{Procedure} Remove2($e, C, D$)
   22. \hspace{2em} Move $\{e\} \setminus Q'_{C,D}$ from $U$ to $G$
   23. \hspace{2em} forall $J \in A(e)$
   24. \hspace{2em} \hspace{2em} Move $J \setminus Q'_{C,D}$ from $U$ to $G$

**Definition 5.** Given a configuration $C \subseteq E$ and a set of events $D \subseteq E$, we define \text{Alt}_2(C, D) as the set

$$\text{Alt}_2(C, D) := \left( \bigcup_{e \in D} A(e) \right) \cap \text{Alt}(C, D)$$
We define the set of events kept by algorithm 2 at any time as

\[ Q'_{C,D} := C \cup D \cup \bigcup_{e \in C \cup D, J \in \text{conf}(e)} J. \]

FIXME explain

**Theorem 6 (Termination, optimality, completeness).** Algorithm 2 always terminates and calls exactly once function \text{Explore2}(C,\cdot,\cdot) for every maximal configuration \( C \).

FIXME talk about subset optimization

### 6 Improvements

#### 6.1 Prefix Extensions

- polynomial under assumption - but NP-complete in general!

#### 6.2 State Caching

- delaying the ‘garbage collection’ of removed events

#### 6.3 Non-acyclic State Spaces

FIXME combining state matching (cutoffs) with optimality is complex; explain why existing approaches based on statefull exploration and unoptimal (static) POR would be difficult in this setting of optimal DPOR; here or in related work

FIXME intuition about feasible and cutoffs, how to stop FIXME modifications to the algorithm

**Definition 6 (Complete prefix [?]).** An unfolding prefix \( \mathcal{P} := (\hat{E}, \hat{z}, \hat{\#}, h) \) of \( M \) and a set \( \mathcal{K} \subseteq \hat{E} \) of cutoff events in \( \mathcal{P} \), we say that \( \mathcal{P} \) is complete iff

- For every reachable state \( s \in \text{reach}(M) \), there is a configuration \( C \in \text{conf}(\mathcal{P}) \) with \( C \cap \mathcal{K} = \emptyset \) and \( \text{state}(C) = s \).
- For any configuration \( C \in \text{conf}(\mathcal{P}) \) such that \( C \cap \mathcal{K} = \emptyset \), if there exists some event \( e \in E \) with \( e \notin C \) and \( C \cup \{ e \} \in \text{conf}(U_M) \), then \( e \in \hat{E} \).

**Definition 7.** An event \( e \in E \) is a causal cutoff iff there is some other event \( e' \in E \) such that \( e' < e \) and \( \text{state}([e]) = \text{state}([e']) \).

We denote by \( \mathcal{K}_c \) the set of causal cutoffs. It is well known that causal cutoffs define a finite and complete prefix:

- Define a variant of \( a1 \) as complete if for every reachable state \( s \in \text{reach}(M) \), the algorithm explores at least one configuration \( C \) such that for some \( C' \subseteq C \) it holds that \( \text{state}(C') = s \).

**Theorem 7 (Completeness).** Algorithm 1 updated with the cutoff mechanism described above is complete.
7 Experiments

[TODO: ...]

8 Related Work

Applying DPOR to programs with cyclic state spaces require the usage of provisos [JU92] which ensures that the subset of transitions explored at a state do not generate a successor state that is present in the stack of the DFS search. As noted in [NG02], SPIN requires a stronger condition where no selected transition at a state should generate a successor state in the stack. In other words, all reached states have to be maintained for the cycle check. This leads to relatively higher space complexity for the DPOR algorithms.

ODPOR [AAJS14] is a closely related work which operates on an interleaved execution model and explores only one execution per Mazurkiewicz trace. However, in order to do so the ODPOR algorithm maintains a data structure (known as —em wake-up trees) to avoid partial explorations which may not eventually reverse a race. In short, wake-up trees store chain of dependencies that must be executed in order to explore a certain deadlock state. It is important to note that constructing and managing wake-up trees can be expensive. Wake-up trees can be potentially as large as the program and inserting an instruction sequence “w” in the wake-up tree involves at least $|v| \times |w|!$ many comparisons where $v$ is the smallest sequence in the wake-up tree such that starting an execution from $v$ is equivalent to starting an execution from $w$.

[TODO: In [KSH12] they employ unfoldings as a ‘tool’ to discover paths of the program, not as a semantics. Make this clear]

9 Conclusions

[TODO: ...]

References


A Proofs: Unfolding Semantics

Lemma 1. [César: Marcelo, start writing a proof for this lemma, should be simple] For any unfolding prefix \((E, <, \#), h)\) we have the following:

1. The relation \(<\) is a strict partial order.
2. The relation \(\#\) is irreflexive.

Proof. We prove both statements separately.

1. Clearly \(e < e\) does not hold, as every event introduced by Definition 2 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.

2. By contradiction. Assume that \(e \neq e\) and that \(e\) has been inserted in the unfolding prefix \((E, <, \#)\) by applying Definition 2 on the unfolding prefix \(P\). Clearly, \(e \not\in D_{P,e}\), so the conflict has not been inserted when extending \(P\) with \(e\). It must be the case, then, that Definition 2 has inserted another event \(e'\) in \(E\) after inserting \(e\), and that \(e' \in [e]\). This is also not possible since, by definition, when inserting \(e'\) on a prefix \(P'\) no causal successor of \(e'\) can be present in \(D_{P',e'}\).

Lemma 2. [César: Marcelo, then prove this one. Observe that \(C\) is now a possibly infinite configuration, the proof I wrote was for \(C\) being finite.] For any configuration \(C\) of an unfolding prefix, \(\text{inter}(C) \subseteq \text{runs}(M)\). Furthermore, if \(C\) is finite, then for any \(\sigma_1, \sigma_2 \in \text{inter}(C)\), we have \(\text{state}(\sigma_1) = \text{state}(\sigma_2)\).

Proof. Let \(P := (E, <, \#, h)\) be an unfolding prefix, with \(h: E \to T\). Let \(C\) be a configuration of \(P\). The proof is by structural induction on the set of unfolding prefixes ordered by the prefix relation \(\leq\).

Base case. Assume that \(P\) has been produced by the first rule of Definition 2. Then \(E = \{1\}\) and the lemma is trivial to prove.

Inductive step. Assume \(P\) that has been produced by the application of the second rule of Definition 2 to the unfolding prefix \(P'\), and let \(e\) be the only event in \(P\) but not in \(P'\). Also, assume that the lemma holds for \(P'\).

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

\[\sigma = h(e_1), \ldots, h(e_n)\]

and that \(e_i = e\). Clearly, the causes \([e]\) of \(e\) are a subset of the events \(\{e_1, \ldots, e_{i-1}\}\). Since, by definition of \(\text{inter}(\cdot)\), \(\{e_1, \ldots, e_{i-1}\}\) is a configuration and it does not include \(e\), it is necessarily a configuration of \(P'\). 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 $[e]$ and then all remaining events in $\{e_1, \ldots, e_{i-1}\}$. This means that $\sigma$ 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 $\sigma'' \in \text{inter}([e])$, $\{f_1, \ldots, f_k\} = \{e_1, \ldots, e_{i-1}\} \setminus [e]$, and $g_1 = e_{i+1}$, $\ldots$, $g_l = e_n$.

Now we need to show that $\sigma''.h(f_1)\ldots h(f_k).h(e)$ is an execution. From Definition 2 we know that $\sigma''$ enables $h(e)$, and from the induction hypothesis we also know that $\sigma'$ enables $h(f_1)$. Since $-f_1 \not\in e$ and $f_1 \not\in [e]$, from Definition 2 we know that $h(f_1) \circ h(e)$, i.e., the transitions associated to both events commute (at all states). Since both $h(f_1)$ and $h(e)$ are enabled at $\text{state}(\sigma'')$, 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 $h(e) \circ h(f_2)$, 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 $h(e) \circ h(g_j)$ for $j \in \{1, \ldots, l\}$. Since $\tilde{\sigma}$ enables both $h(e)$ and $h(g_1)$, and both commute at $\text{state}(\tilde{\sigma})$, 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 $\sigma$ is indeed an execution.

The lemma also requires to prove that any two executions in $\text{inter}(C)$ 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 4.** [César: We don’t need this lemma anymore. I just left it here] For any two prefixes $\mathcal{P}_1, \mathcal{P}_2$ resulting from extending a prefix $\mathcal{P}$ with respectively two different events, there is a prefix $\mathcal{P}_3$ with $\mathcal{P}_1 \leq \mathcal{P}_3$ and $\mathcal{P}_2 \leq \mathcal{P}_3$.

**Lemma 4.** [César: We don’t need this lemma anymore. I just left it here] For any two prefixes $\mathcal{P}_1, \mathcal{P}_2$ resulting from extending a prefix $\mathcal{P}$ with respectively two different events, there is a prefix $\mathcal{P}_3$ with $\mathcal{P}_1 \leq \mathcal{P}_3$ and $\mathcal{P}_2 \leq \mathcal{P}_3$.

**Proof.** Assume that $\mathcal{P} := ((E, <, \#), h)$ is a prefix. Let $\mathcal{P}_1 := ((E_1, <_1, \#_1), h_1)$ and $\mathcal{P}_2 := ((E_2, <_2, \#_2), h_2)$ be the prefixes obtained by applying the second rule of Definition 2 to the prefix $\mathcal{P}$. Assume that $\mathcal{P}_1$ is the result of extending $\mathcal{P}$ with
one event \( e_1 := (t_1, H_1) \), and similarly for \( e_2 := (t_2, H_2) \) and \( P_2 \). Clearly both histories \( H_1, H_2 \subseteq E \) are configurations of \( P \). Since the addition of one event \( e \) to the prefix \( P \) yields a prefix that has at least the same histories, as the only new conflicts that are introduced are between \( e \) and other events, never between existing events, it must be the case that \( H_1 \) and \( H_2 \) are also configurations of both \( P_1 \) and \( P_2 \). As a result one can clearly invoke the second rule of Definition 2 and extend \( P_1 \) with \( e_2 \) and \( P_2 \) with \( e_1 \). Let \( P_3 \) and \( P_3' \) be, respectively, the resulting unfolding prefixes.

We now need to verify that \( P_3 \) and \( P_3' \) are indeed the same prefix. Clearly, they contain the same events. Regarding the causality relation, consider any event \( e \) in \( P_3 \). If \( e \neq e_2 \) we have that \( e \prec e_1 \) holds in \( P_3 \) iff it holds in \( P_3' \), as the history of \( e_1 \) is the same in both. If \( e = e_2 \) then clearly \( \neg(e < e_1) \) holds both in \( P_3 \) and \( P_3' \), because \( e \) was not an event of \( P \). This implies that any pair of events causally related in \( P_3 \) are also causally related in \( P_3' \), and a symmetric argument shows the analogous statement in the opposite direction. Thus, both prefixes have the same causality relation.

Now we check that both \( P_3 \) and \( P_3' \) have the same conflict relation. Since \( e_1 \) is the last event added to \( P_3' \), clearly the set of events \( e' \) such that \( e_1 \neq e' \) is exactly \( D_{P_2,e_1} \). We need to show that for any \( e' \in D_{P_2,e_1} \) we can find a conflict between \( e_1 \) and \( e' \) also in \( P_3 \). If \( e' \) is an event of \( P_1 \), then clearly it is contained in \( D_{P,e_1} \), and so it is in conflict with \( e_1 \) in \( P_3 \). It cannot be the case that \( e' = e_1 \), by definition of \( D_{P_2,e_1} \), so the only remaining possibility is that \( e = e_2 \). In this case, we have \( t_1 \otimes t_2 \). Also, we know that both \( \neg(e_1 < e_2) \) and \( \neg(e_2 < e_1) \) hold. This implies that \( e_1 \in D_{P_1,e_2} \), and since \( e_2 \) was the last event added to \( P_1 \), this means that Definition 2 has set \( e_2 \neq e_1 \) in \( P_3 \). This proves that the conflict relation on both \( P_3 \) and \( P_3' \) is also the same.

**Theorem 1.** [César: This is a breakfast-proof corollary of Lemma 3] The unfolding \( U_M \) exists and is unique.

**Proof.** There is at least one unfolding prefix within the set of unfolding prefixes defined by Definition 2. Also, there are finitely many unfolding prefixes because we have assumed that the system has finitely many (finite) executions. The set of unfolding prefixes is therefore finite and non-empty, so some \( \preceq \)-maximal prefix must exist.

To show that it is unique, assume, for a proof by contradiction, that it is not. Let \( P_1 \) and \( P_2 \) be two \( \preceq \)-maximal unfolding prefixes. W.l.o.g., since \( \neg(P_1 \preceq P_2) \) holds, it must be the case that \( P_1 \) contains some event not contained in \( P_2 \). Furthermore, from the non-empty set of events contained in \( P_2 \) but not in \( P_1 \) there must exist some event \( e \) such that \( [e] \) is a configuration of \( P_1 \). By Lemma 4, it must be possible to extend \( P_1 \) with such event, producing another unfolding prefix \( P_1' \) such that \( P_1 \preceq P_1' \). This is a contradiction to the maximality of \( P_1 \).

**Theorem 2.** [César: Finally go for this one, see my comment at the beginning of the current proof] For any non-empty run \( \sigma \) of \( M \), there exists a unique configuration \( C \) of \( U_M \) such that \( \sigma \in \text{inter}(C) \).
Proof. [César: Marcelo, it is straight-forward to modify this proof so that this theorem will be able to handle the case where \( \sigma \) is infinite. You first prove the theorem assuming that \( \sigma \) is finite, i.e., you reuse (as is?) the current proof.

Then you deal with infinite \( \sigma \). You define a function that associates \( \sigma \) to a set of events constructed as a union of infinitely many configurations, one for each finite prefix of \( \sigma \). Then you prove that that is a configuration. To show uniqueness, by contradiction should be quite easy. Good luck!]

Observe that for a run that fires no transition, i.e. \( \sigma = \varepsilon \in T^* \), we may find the empty configuration \( \emptyset \) or the configuration \( \{ 1 \} \), and in both cases \( \sigma \) is an interleaving of the configuration. Hence the restriction to non-empty runs.

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

**Base Case.** If \( \sigma \) fires one transition \( t \), then \( t \) is enabled at \( s \), the initial state of \( M \). Then \( \{ 1 \} \) is a history for \( t \), as necessarily state(\( \{ 1 \} \)) enables \( t \). This means that \( e := (t, \{ 1 \}) \) is an event of \( \mathcal{U}_M \), and clearly \( e \in \text{inter}(\{ 1 \}, e) \). It is easy to see that no other event \( e' \) different than \( e \) but such that \( h(e) = h(e') \) can exist in \( \mathcal{U}_M \) and verify that history \([e']\) of \( e' \) is the singleton \( \{ 1 \} \). The representative configuration for \( \sigma \) is therefore unique.

**Inductive Step.** Consider \( \sigma = \sigma'.t_{k+1}, \) with \( \sigma' = t_1. t_2 \ldots t_k \). By the induction hypothesis, we assume that there exist a unique configuration \( C' \) such that \( \sigma' \in \text{inter}(C') \). By Lemma 2, all runs in \( \text{inter}(C') \) reach the same state \( s \) and \( \sigma' \) is such a run. Hence, \( t_{k+1} \) is enabled at state \( s \). If all events \( e \in \text{max}(C'): h(e) \) interfere with \( t_{k+1} \), then \( C' \) is a valid configuration \( H \) and by construction (second condition of Definition 2) there is a configuration \( C = C' \cup \{ e' \} \) with \( e' = (t_{k+1}, H) \). Otherwise, we construct a valid \( H \) by considering sub-configurations of \( C' \) removing a maximal event \( e \in \text{max}(C'): h(e) \) does not interfere with \( t_{k+1} \). We always reach a valid \( H \) since \( C' \) is a finite set and \( \{ 1 \} \) is always a valid \( H \).

Considering \( C = H \cup \{ e' \} \) with \( e' = (t, H) \), by construction (second condition of Definition 2) we have that \( \forall e_H \in H : \neg (e' \# e_H) \) and \( \forall e_R \in C' \setminus H : \neg (e \# e_R) \) (otherwise these events would be in \( H \)). Hence, \( C' \cup \{ e \} \) is a configuration.

B Proofs: Lazy Exploration Algorithm

[TODO: Review all proofs; definitions of \texttt{Alt} has changed]

Algorithm 1 is recursive, each call to \texttt{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 \( \text{en}(C) \) or the set \( A \cap \text{en}(C) \), 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 algorithm 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 $(C, D, A, e)$, where $C, D, A$ are the parameters of a recursive call made to the function $\text{Explore}(\cdot, \cdot, \cdot)$, and $e$ is the event selected by the algorithm immediately before line 9. More formally, $B$ contains exactly all tuples $(C, D, A, e)$ satisfying that

- $C$, $D$, and $A$ are sets of events of the unfolding $U_M$;
- during the execution of $\text{Explore}(\emptyset, \emptyset, \emptyset)$, the function $\text{Explore}(\cdot, \cdot, \cdot)$ has been recursively called with $C, D, A$ as, respectively, first, second, and third argument;
- $e \in E$ is the event selected by $\text{Explore}(C, D, A)$ immediately before line 9 if $C$ is not maximal; if $C$ is maximal, we define $e := 1$.\(^3\)

The edge relation of the call graph, $\triangleright \in B \times B$, represents the recursive calls made by $\text{Explore}(\cdot, \cdot, \cdot)$. Formally, it is the union of two disjoint relations $\triangleright := \triangleright_1 \cup \triangleright_r$, defined as follows. We define that

$$(C, D, A, e) \triangleright_1 (C', D', A', e') \quad \text{and} \quad (C, D, A, e) \triangleright_r (C'', D'', A'', e'')$$

iff the execution of $\text{Explore}(C, D, A)$ issues a recursive call to, respectively, $\text{Explore}(C', D', A')$ at line 9 and $\text{Explore}(C'', D'', A'')$ 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.

Observe that $(B, \triangleright)$ is by definition a weakly connected digraph, as there is a path from the node $(\emptyset, \emptyset, \emptyset, \cdot)$ to every other node in $B$. Later in this section we will additionally prove that the call graph is actually a binary tree, where $\triangleright_1$ is the left-child relation and $\triangleright_r$ is the right child relation.

## B.1 General Lemmas

**Lemma 5.** Let $(C, D, A, e) \in B$ be a state of the call graph. We have that

- event $e$ is such that $e \in \text{en}(C)$; \hspace{1cm} (7)
- $C$ is a configuration; \hspace{1cm} (8)
- $C \cup A$ is a configuration and $C \cap A = \emptyset$; \hspace{1cm} (9)
- $D \subseteq \text{ex}(C)$; \hspace{1cm} (10)
- if $A = \emptyset$, then $D \subseteq \text{ex}(C)$; \hspace{1cm} (11)
- for all $e' \in D$ there is some $e'' \in C \cup A$ such that $e' \not\sim e''$ \hspace{1cm} (12)

**Proof.** To show (7) is immediate, as in both branches of the preceding conditional statement, event $e$ is chosen from the set $\text{en}(C)$.

All remaining items, (8) to (12), 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 $b_0 := (\emptyset, \emptyset, \emptyset, e_0)$ and leading to $b_n := (C, D, A, e)$ (we will later show that there is actually only one such path). For $i \in \{0, \ldots, n\}$ we define $(C_i, D_i, A_i, e_i) := b_i$.

\(^3\) Observe that in this case, if $C$ is maximal, the execution of $\text{Explore}(C, D, A)$ never reaches line 9.
We start showing (8). **Base case.** $n = 0$ and $C = \emptyset$. The empty set is a configuration. **Step.** Assume $C_{n-1}$ is a configuration. If $b_{n-1} \triangleright_I b_n$, then $C = C_{n-1} \cup \{e\}$ for some event $e \in \text{en}(C)$, as stated in (7). 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 now show (9), also by induction on $n$. **Base case.** $n = 0$. Then $C = A = \emptyset$ and clearly $C \cup A$ is a configuration and $C \cup A = \emptyset$. **Step.** Assume that $C_{n-1} \cup A_{n-1}$ is a configuration and that $C_{n-1} \cap A_{n-1} = \emptyset$. We have two cases.

- Assume that $b_{n-1} \triangleright_I b_n$. 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 $b_{n-1} \triangleright_R b_n$. Then $C = C_{n-1}$, and $A = [J] \setminus C_{n-1}$ for some $J \in \mathcal{A} \triangleright_I (C_{n-1}, D)$. 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 $C \cap A = \emptyset$.

We show (10), again, by induction on $n$. **Base case.** $n = 0$ and $D = \emptyset$. Then (10) clearly holds. **Step.** Assume (10) holds for $(C_i, D_i, A_i, e_i)$, with $i \in \{0, \ldots, n-1\}$. We show that it holds for $b_n$. As before, we have two cases.

- Assume that $b_{n-1} \triangleright_I b_n$. 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 $[e'] \subseteq C$ and $e' \notin C$. By induction hypothesis we know that $D = D_{n-1} \subseteq \text{ex}(C_{n-1})$, so clearly $[e'] \subseteq C_{n-1} \subseteq C$.

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 (12) we would have that some event in $C$ is in conflict with some other event in $C \cup A$, which is a contradiction to (9).

- Assume that $b_{n-1} \triangleright_R b_n$. We have that $D = D_{n-1} \cup \{e_{n-1}\}$, and by hypothesis we know that $D_{n-1} \subseteq \text{ex}(C_{n-1}) = \text{ex}(C)$. As for $e_{n-1}$, by (7) we know that $e_{n-1} \in \text{en}(C_{n-1}) = \text{en}(C) \subseteq \text{ex}(C)$. As a result, $D \subseteq \text{ex}(C)$.

We show (11). By (10) we know that $D \subseteq \text{ex}(C)$. Assume $A = \emptyset$. For each $e' \in D$ we need to prove the existence of some $e'' \in C$ with $e' \neq e''$. This is exactly what (12) states.

We show (12), again, by induction on $n$. **Base case.** $n = 0$ and $D = \emptyset$. The result holds. **Step.** Assume (12) holds for $(C_{n-1}, D_{n-1}, A_{n-1}, e_{n-1})$. We show that it holds for $b_n$. We distinguish two cases.

- $b_{n-1} \triangleright_I b_n$. 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 $e' \neq e''$. 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.
Lemma 6. If \( C \subseteq C' \) are two finite configurations, then \( \text{en}(C) \cap (C' \setminus C) = \emptyset \)
iff \( C' \setminus C = \emptyset \).

**Proof.** If there is some \( e \in \text{en}(C) \cap (C' \setminus C) \), 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, \( [e''] \subseteq C \). Since \( e'' \notin C \) and \( C \cup \{e''\} \) is a configuration (as \( C \cup \{e''\} \subseteq C' \)), we have that \( e'' \in \text{en}(C) \). Then \( \text{en}(C) \cap (C' \setminus C) \) is not empty.

Lemma 7. For any node \( (C, D, A, e) \in N \) of the call graph we have that \( A \neq \emptyset \)
implies \( \text{en}(C) \cap A \neq \emptyset \).

**Proof.** The result is a consequence of Lemma 6 and (9). Since \( C \cup A \) is configuration that includes \( C \), and \( (C \cup A) \setminus C = A \) is not empty, then \( \text{en}(C) \cap A \) is not empty.

Lemma 8. Let \( b := (C, D, A, e) \) and \( b' := (C', D', A', e') \) be two nodes of the call graph such that \( b \triangleright b' \). Then

- \( C \subseteq C' \) and \( D \subseteq D' \);
- if \( b \triangleright_I b' \), then \( C \not\subseteq C' \);
- if \( b \triangleright_r b' \), then \( D \not\subseteq D' \).

**Proof.** If \( b \triangleright_I b' \), then \( C' = C \cup \{e\} \) and \( D' = D \). Then all the three statements hold. If \( b \triangleright_r b' \), then \( C' = C \) and \( D' = D \cup \{e\} \). Similarly, all the three statements hold.

[César: introduce the proposition, saying that it is valuable alone]

Proposition 1. Let \( b := (C, D, A, e) \in B \) be a node of the call graph and \( J \in \text{Alt}(C, D \cup \{e\}) \) an arbitrary alternative to \( D \cup \{e\} \) after \( C \), of the form

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

such that for all \( i \in \{1, \ldots, n\} \) there is some \( \hat{e} \in D \cup \{e\} \) with \( \hat{e} \in \text{k}(e_i) \).

Then there is a node \( b' := (C', D', A', e') \in B \) satisfying \( b \triangleleft^* b' \) and \( J \subseteq C' \).
Proof. [César: This lemma should say, if $J$ is not taken, then it is in the alternatives my my right child] [César: And go to the general lemmas?] Since the set of alternatives to $D \cup \{e\}$ after $C$ is not empty, it contains $J$, there is a node $b_1 := (C_1, D_1, A_i, \tilde{e}_i) \in B$ such that $b \triangleright_r b_1$. Consider the path made by algorithm 1 in the call graph to explore the leftmost branch of the subtree rooted at $b_1$, namely

$$b_1 \triangleright_1 b_2 \triangleright_1 \ldots \triangleright_1 b_m,$$

with $b_m$ being a leaf node (maximal configuration). Let $(C_i, D_i, A_i, \tilde{e}_i) := b_i$ for $i \in \{1, \ldots, m\}$.

If $J \subseteq C_m$ we are done, as we can take $b' := b_m$. If $J \notin C_m$, since $C_m$ is a maximal configuration, there is some unique $i \in \{1, \ldots, m-1\}$ and some $\hat{e} \in J$ such that

- $\hat{e} \# \tilde{e}_i$,
- for $j \in \{1, \ldots, i\}$ it holds that $C_j \cup J$ is a configuration.

We can assume, w.l.o.g., that $\hat{e} \#^i \tilde{e}_i$.

We will show now that $J$ is an alternative to $D_i \cup \{\tilde{e}_i\}$ after $C_i$, i.e. $J \in \text{Alt}(C_i, D_i \cup \{\tilde{e}_i\})$. To do so, we need to prove that (5) and (6) hold. By definition $C_i \cup J$ is a configuration, so (5) holds. To show (6) we can show that for every $\hat{e} \in D_i \cup \{\tilde{e}_i\}$ we have some $\hat{e} \in C_i \cup J$ with $\hat{e} \in \mathcal{K}(\hat{e})$. Remark that $D_i = D \cup \{e\}$, since the path from $b$ to $b_i$ went right only once.

- If $\hat{e} \in D \cup \{e\}$, since $J$ is an alternative to $D \cup \{e\}$ after $C$, there is some $\hat{e} \in C \cup J$ with $\hat{e} \in \mathcal{K}(\hat{e})$. But, by (13), $C \subseteq C_i$, so $\hat{e} \in C_i \cup J$.
- If $\hat{e} = e_i$ then we know by definition of the index $i$ that there is some $\hat{e} \in J$ with $\hat{e} \#^i e_i$. We need to show that $\hat{e} \in \mathcal{K}(e_i)$, i.e., that $\hat{e}$ will be remembered by the time $e_i$ is considered for addition to $C_i$, even if $\hat{e} \notin C_i \cup A_i$. Recall that $J$ has the form $J = \{e_1, \ldots, e_n\}$, and each $e_i$, with $i \in \{1, \ldots, n\}$, belongs to $\mathcal{K}(e')$, for some $e' \in D \cup \{e\}$. Then necessarily there is some $j \in \{1, \ldots, n\}$ such that $\hat{e} < e_j$, and then, since $e_j$ is remembered, $\hat{e}$ must be remembered as well. Since $\hat{e}$ is remembered when $e_i$ is discovered, it will be present in $\mathcal{K}(e_i)$ when the algorithm evaluates $\text{Alt}(C_i, D_i \cup \{\tilde{e}_i\})$.

In any case, (13) holds for $J$, and $J \in \text{Alt}(C_i, D_i \cup \{\tilde{e}_i\})$.

Now, since $J$ satisfies the hypothesis of this lemma on $b_i$ we can repeat exactly the same argument starting from $b_i$ rather than $b$. Observe that we did progress from $b$ to $b_i$, i.e., algorithm 1 did at least one recursive call to move from $b$ to $b_i$. And remark that $b \triangleright^* b_i$. Since, by Lemma 9, all paths from $b$ are finite, and we did progress, we will only need to re-apply the argument a finite number of times until the leftmost leaf $b'$ of the tree rooted at $b_1$ contains $J$. This is what we wanted to prove.

B.2 Termination

Lemma 9. Any path $b_0 \triangleright b_1 \triangleright b_2 \triangleright \ldots$ in the call graph starting from $b_0 := (\emptyset, \emptyset, \emptyset, \bot)$ is finite.
Proof. By contradiction. Assume that $b_0 \triangleright b_1 \triangleright \ldots$ is an infinite path in the call graph. For $0 \leq i$, let $(C_i, D_i, A_i, e_i) := b_i$. Recall that $\mathcal{U}_M$ 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 $\triangleright_l$ rather than $\triangleright_r$ is finite, since every time $\text{Explore}(\cdot, \cdot, \cdot)$ makes a recursive call at line 9 it adds one event to $C_i$, as stated by (14). More formally, the set

$$L := \{i \in \mathbb{N} : C_i \triangleright_l C_{i+1}\}$$

is finite. As a result it has a maximum, and its successor $k := 1 + \max < L$ is an index in the path such that for all $i \geq k$ we have $C_i \triangleright_r C_{i+1}$, i.e., the function only makes recursive calls at line 11. We then have that $C_i = C_k$, for $i \geq k$, and by (10), that $D_i \subseteq \text{ex}(C_k)$. Recall that $\text{ex}(C_k)$ is finite. Observe that, due to (14), the sequence

$$D_k \nsubseteq D_{k+1} \nsubseteq D_{k+2} \nsubseteq \ldots$$

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

**Corollary 1.** The call graph is a finite directed acyclic graph.

**Proof.** Recall that every node $b \in B$ is reachable from the initial node $b_0 := (\emptyset, \emptyset, \emptyset, \bot)$, by definition of the graph. Also, by Lemma 9, all paths from $b_0$ are finite, and every node has between 0 and 2 adjacent nodes.

By contradiction, if the graph had infinitely many nodes, then König’s lemma would guarantee the existence of an infinite path starting from $b_0$, a contradiction to Lemma 9. Then $B$ is necessarily finite.

As for the acyclicity, again by contradiction, assume that $(B, \triangleright)$ has a cycle. The every state of one such cycles 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 9.

**Theorem 3 (Termination).** Regardless of its input, algorithm 1 always stops.

**Proof.** Remark that algorithm 1 makes calls to three functions, namely, $\text{Extend}(\cdot)$, $\text{Remove}(\cdot)$, and $\text{Alt}(\cdot, \cdot)$. Clearly the first two terminate. Since we gave no algorithm to compute $\text{Alt}(\cdot)$, we will assume we employ one that terminates on every input.

[\text{César: fix this, there is no more loop there}] Now, observe that there is only one loop in algorithm 1, at ??, which clearly will iterate for no more than $|C| \cdot \text{ex}(C)$ times, for any configuration $C$. Thus any non-terminating execution of algorithm 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 9, no infinite path exist in the call graph, algorithm 1 always terminates.
B.3 Optimality

Lemma 10. Let \( b, b_1, b_2, b_3, b_4 \in B \) be nodes of the call graph such that
\[
b \triangleright_i b_1 \triangleright^* b_3 \quad \text{and} \quad b \triangleright_r b_2 \triangleright^* b_4.
\]
and such that \( (C_3, D_3, A_3, e_3) := b_3 \) and \( (C_4, D_4, A_4, e_4) := b_4 \). 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 (14) we know that \( e \in C_1 \), and by (13) that \( e \in C_3 \). We show that \( e \notin C_4 \). By (15) we have that \( e \in D_2 \), and again by (13) that \( e \in D_4 \). Since \( D_4 \subseteq \text{ex}(C_4) \), by (10), we have that \( e \in \text{ex}(C) \), so \( e \notin C_4 \).

Corollary 2. The call graph \( (B, \triangleright) \) is a finite binary tree, where \( \triangleright_i \) and \( \triangleright_r \) are respectively the left-child and right-child relations.

Proof. Corollary 1 states that the call graph is a finite directed acyclic graph. Lemma 10 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 11. For any maximal configuration \( C \subseteq E \), there is at most one node \( \langle C, D, A, \hat{e} \rangle \in B \) with \( C = \hat{C} \).

Proof. By contradiction, assume there was two different nodes,
\[
\hat{b} := \langle C, D, \hat{A}, \hat{e} \rangle \quad \text{and} \quad b' := \langle C, D', A', e' \rangle
\]
in \( B \) such that the first component of the tuple is \( C \). The call graph is a binary tree, due to Corollary 2, so there is exactly one path from \( b_0 := \langle \emptyset, \emptyset, \emptyset, e_0 \rangle \) to respectively \( \hat{b} \) and \( b' \). Let
\[
\hat{b}_0 \triangleright \hat{b}_1 \triangleright \ldots \triangleright \hat{b}_{n-1} \triangleright \hat{b}_n \quad \text{and} \quad b'_0 \triangleright b'_1 \triangleright \ldots \triangleright b'_{m-1} \triangleright b'_m
\]
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 \( \hat{b} \triangleright b'' \) or \( b' \triangleright b'' \). 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 \( \hat{b}_i \triangleright_i \hat{b}_{i+1} \) and that \( b'_i \triangleright_r b'_{i+1} \). Now, using (14) and (13), it is simple to show that \( e_i \in C \). And using (15) and (13), that \( e_i \in D' \). Then, by (10) we get that \( e_i \in \text{ex}(C) \), a contradiction to \( e_i \in C \).
Lemma 11. There is at most one node with its first parameter being equal to $\tilde{C}$.

Proof. By construction, every call to $\text{Explore}(C, D, A)$ produces one node of the form $(C, D, A, e)$, for some $e \in E$, in the call graph associated to the execution. By Lemma 11, there is at most one node with its first parameter being equal to $\tilde{C}$, so $\text{Explore}(\cdot, \cdot, \cdot)$ can have been called at most once with $\tilde{C}$ as first parameter.

### B.4 Completeness

**Lemma 12.** Let $b := (C, D, A, e) \in B$ be a node in the call graph and $\tilde{C} \subseteq E$ an arbitrary maximal configuration of $\mathcal{U}_M$ such that $C \subseteq \tilde{C}$ and $D \cap \tilde{C} = \emptyset$. Then exactly one of the following statements holds:

- Either $C$ is a maximal configuration of $\mathcal{U}_M$, or
- $e \in \tilde{C}$ and $b$ has a left child, or
- $e \notin \tilde{C}$ 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 $(B, \succ)$ is a binary tree (Corollary 2). We let $< \subseteq B \times B$ be the unique in-order relation in $B$. More formally, we define $<$ to be the order that sorts, for every $b \in B$, first all nodes reachable from $b$'s left child (if there is any), then $b$, then all nodes reachable from $b$'s right child (if there is any).

**Base case.** Node $b$ is the least element in $B$ w.r.t. $<$. Then $b$ is the leftmost leaf of the call tree, i.e., $b_0 \succ^* b$, with $b_0 := (\emptyset, \emptyset, \emptyset, \emptyset)$ and $C$ is a maximal configuration. Then the first item holds.

**Step case.** Assume that the result holds for any node $\tilde{b} < b$. 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 $e \in \tilde{C}$ then we are done, as the second item holds.

So assume that $e \notin \tilde{C}$. 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 $\hat{J} \subseteq \tilde{C}$ such that $\hat{J} \in \text{Alt}(C, D \cup \{e\})$.

We start by setting up some notation. Observe that any alternative $J \in \text{Alt}(C, D \cup \{e\})$ needs to contain, for every event $e' \in D \cup \{e\}$, some event $e'' \in J \cap C$ 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

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

Let $e_i$ be any event in $F$. Clearly $e_i \in \text{ex}(\hat{C})$, as $e_i \in D \subseteq \text{ex}(C)$, by (10), and so $[e_i] \subseteq C \subseteq \hat{C}$ and $e_i \notin \tilde{C}$. Since $e_i \in \text{ex}(\hat{C})$ 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 $e_i \neq e_i'$ for $i \in \{1, \ldots, n\}$. Clearly $\hat{J} \in \hat{C}$ 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 $\hat{J} \in \text{Alt}(C, D \cup \{e\})$ when function $\text{Alt}(\cdot)$ is called just before line 11 during the execution of $\text{Explore}(C, D, A)$. Let $\hat{U}$ be the set of events contained in variable $U$ of algorithm 1 exactly when $\text{Alt}(\cdot)$ is called. Clearly $C \cup \hat{J}$ is a configuration, so (6) holds. To verify (5), consider any event $\hat{e} \in D \cup \{e\}$. If $\hat{e} \in D \cap \text{cex}(C)$ we can always find some $\hat{e}' \in C$ with $\hat{e}' \neq \#_1(\hat{e})$.

If not, then $\hat{e} = e_i$ for some $i \in \{1, \ldots, n\}$ and we can find some $e_i' \in \hat{J}$ such that $e_i \neq e_i'$. 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 $\text{Alt}(\cdot)$ 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 $\text{Alt}(C, D \cup \{e\})$ is called. The set $U$ has been filled with events in function $\text{Extend}(\cdot)$ as the exploration of $U_M$ advanced, some of them have been kept in $U$, some of them have been removed with $\text{Remove}(\cdot)$. To reason about the events in $\hat{U}$ we need to look at fragment of $\mathcal{U}\chi$ 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

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

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 $\text{Explore}(C_i, D_i, A_i)$ is right now blocked on the right-hand side recursive call at line 11 in algorithm 1, after having decided that there was one right child to explore. For the sake of clarity, we can then informally write

$$b_0 \triangleright^* b_1 \triangleright_r \triangleright^* b_2 \triangleright_r \triangleright^* \ldots \triangleright_r \triangleright^* b_n.$$

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 $\text{Explore}(C_i, D_i, A_i)$ just before the right recursive call at line 11 was made, i.e., the value of variable $U$ when $\text{Alt}(C_i, D_i \cup \{e_i\})$ was called. For $i = 0$ we set $U_0 := \{\_\}$ 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 $\hat{b} := (\hat{C}, \hat{D}, \cdot, \hat{e}) \in B$ explored after $b_i$ and before $b_n$ it holds that $e_i \in \hat{D}$, by (13), and so every time function $\text{Remove}(\hat{e}, \hat{C}, \hat{D})$ 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:

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

First, note that $C'$ is indeed a configuration, since it is clearly causally closed and there is no conflict: $e_i \in \text{en}(C)$ and $C \cup \{e'_i\} \subseteq \hat{C}$ and $[e_i] \cup \{e'_i\}$ is conflict-free (because $e_i$ and $e'_i$ are in immediate conflict). Remark also that $D_i \subseteq \text{ex}(C')$ and that $e'_i \in \text{cex}(C')$. We now regard two cases:

- **Case 1**: there is some maximal configuration $C'' \supseteq C'$ such that $D_i \cap C'' = \emptyset$. We show that $C''$ has been visited during the exploration of the left subtree of $b_i$. In that case, since $e_i' \in \text{cex}(C'')$ and $e_i \in C''$, algorithm 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 $b_i' := (C_i \cup \{e_i\}, D_i, \cdot)$ of $b_i$. In that case, since $b_i < b$ (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., $b_i' > b$) is such that $\hat{b} < b_i < b$. This means that the induction hypothesis applies to $\hat{b}$. So Lemma 13 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 $D_i \cap C'' = \emptyset$. In other words, any maximal configuration $C'' \supseteq C'$ is such that $D_i \cap C'' \neq \emptyset$.

Our first step is showing that this implies that

$$\exists j \in \{1, \ldots, i-1\} \text{ such that } \#(e_i) \cap \hat{C} \supseteq \#(e_j) \cap \hat{C}. \quad (16)$$

Let $C'' \supseteq C'$ be a maximal configuration. Then $D_i \cap C'' \neq \emptyset$. This implies that $D_i \cap \text{en}(C) \cap C'' \neq \emptyset$, as necessarily $D_i \cap C'' \nsubseteq \text{en}(C)$. Observe that $D_i \cap \text{en}(C) = \{e_1, \ldots, e_{i-1}\}$, so we have that $\{e_1, \ldots, e_{i-1}\} \cap C'' \neq \emptyset$. 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 $D_i \cap C''' = \emptyset$ (for instance, take $C''' := \hat{C}$). But, because $C' \subseteq X_2$, we cannot find any $C''' \supseteq X_2$ satisfying that $D_i \cap C''' = \emptyset$. This implies that for any $C''' \supseteq X_2$ we have $\{e_1, \ldots, e_{i-1}\} \cap C''' \neq \emptyset$. Based on
the last statement we can now prove (16) by contradiction. Assume that
(16) 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 would mean that for any \( j \in \{1, \ldots, i - 1\} \) it
holds that \( \#(e_j) \cap X_2 \neq \emptyset \). This implies that any maximal configuration \( C'' \)
extending \( X_2 \) is such that \( \{e_1, \ldots, e_{i-1}\} \cap C'' = \emptyset \). This is a contradiction,
so the validity of (16) is now established.

According to (16) 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:

\[
X_3 := X_1 \cup \{e_m\} \cup [e'_m].
\]

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
  \( X_1 \cup [e'_m] \) is causally closed. Now, since \( \{e_i, e_m\} \subseteq en(C) \), we have that
  \( \#(e_i) \cap C = \emptyset \), and as a result \( [e_m] \subseteq C \subseteq X_1 \subseteq X_3 \).

- **Fact 2:** set \( X_3 \) is conflict free. Since \( X_1 \cup [e'_m] \subseteq \hat{C} \), there is no pair of
  conflicting events in \( X_1 \cup [e'_m] \). Consider now \( e_m \). Since \( e_m \) and \( e'_m \) are in
  immediate conflict, by definition \( e_m \) has no conflict with any event in
  \([e'_m]\). Consider now any event \( \tilde{e} \in X_1 \). Observe that \( \tilde{e} \in \hat{C} \). If \( \tilde{e} \in \#(e_m) \)
  then by (16) we have that \( \tilde{e} \notin \#(e_i) \), 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 (14), 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 \( X_3 \cap D_m = \emptyset \). By (10) and (13) 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 \( j \in \{1, \ldots, m - 1\} \), due to 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 \( X_3 \cap D_m = \emptyset \).

To show that \( X_3 \) has been explored in the subtree rooted at \( b_m \), consider
the left child \( b'_m := (C_m \cup \{e_m\}, D_m, \cdot) \) of \( b_m \). The induction hypothesis
applies to any node \( \hat{b} \in B \) in the subtree rooted at \( b'_m \) (i.e., \( b'_m \triangleright^* \hat{b} \)). This is
because \( \hat{b} < b'_m < b_m < b \). 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 13,
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 \( e_m \not\equiv 3 e'_m \) and \( e_m \) is
in set $D$ we also know that $\text{Remove}(\cdot)$ 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 (16), 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 $e_i \not\#^i \tilde{e}$. 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 $\tilde{C}$ (as we said before, to keep things simple). But we can always find at least one event in $\tilde{C}$ 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 $\hat{J} \in \text{Alt}(C, D \cup \{e\})$ when the set of events $U$ present in memory equals $\hat{U}$. As a result, algorithm 1 will do a recursive call at line 11 and $b$ will have a right child. This is what we wanted to prove.

**Lemma 13.** For any node $b := (C, D, e) \in B$ in the call graph and any maximal configuration $\hat{C} \not\subseteq E$ of $\mathcal{U}_M$, if $C \subseteq \hat{C}$ and $D \cap \hat{C} = \emptyset$ and Lemma 12 holds on all nodes in the subtree rooted at $b$, then there is a node $b' := (C', \cdot, \cdot) \in B$ such that $b \not\triangleright^* b'$, and $\hat{C} = C'$.

**Proof.** Assume that Lemma 12 holds on any node $b'' \in B$ such that $b \not\triangleright^* b''$, i.e., all nodes in the subtree rooted at $b$. Since $C \subseteq \hat{C}$ and $D \cap \hat{C} = \emptyset$, we can apply Lemma 12 to $b$ and $\hat{C}$. If $C$ is maximal, then clearly $C = \hat{C}$ and we are done. If not we regard two cases. If $e \in \hat{C}$ then by Lemma 12 we know that $b$ has a left child $b_1 := (C_1, D_1, \cdot, e_1)$, with $C_1 = C \cup \{e\}$ and $D_1 = D$. Finally, if $e \notin \hat{C}$ then equally by Lemma 12 we know that $b$ has a right child $b_1 := (C_1, D_1, \cdot, e_1)$, with $C_1 = C$ and $D_1 = D \cup \{e\}$. Observe, in any case, that $C_1 \not\subseteq \hat{C}$ and $D_1 \cap \hat{C} = \emptyset$.

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 12 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 $b_n := (C_n, D_n, \cdot, \cdot)$ 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 $C_n \not\subseteq \hat{C}$. This implies that $C_n = \hat{C}$, and we can take $b' := b_n$.

**Theorem 5 (Completeness).** Let $\hat{C}$ be a maximal configuration of $\mathcal{U}_M$. Then $\text{Explore}(\cdot, \cdot, \cdot)$ is called at least once with its first parameter being equal to $\hat{C}$.

**Proof.** We need to show that for every maximal configuration $\hat{C} \subseteq E$ we can find a node $b := (C, \cdot, \cdot, \cdot) \in B$ in the call three such that $\hat{C} = C$. This is a direct
consequence of Lemma 13. Consider the root node of the tree, \( b_0 := (C, D, A, 1) \), where \( C = D = A = \emptyset \). Clearly \( C \subseteq \hat{C} \) and \( D \cap \hat{C} = \emptyset \), and Lemma 12 holds on all nodes of the call tree. So Lemma 13 applies to \( \hat{C} \) and \( b_0 \), and it establish the existence of the aforementioned node \( b \).

B.5 Memory Consumption

**Lemma 14.** Assume the function \( \text{Explore}(C, D, A) \) is eventually called. Let \( \hat{U} \) and \( \hat{U} \) be, respectively, the values of set \( U \) in algorithm 1 immediately before and immediately after executing the call. If \( Q_{C, D, \hat{U}} \subseteq \hat{U} \subseteq Q_{C, D, \hat{U}} \cup en(C) \), then \( \hat{U} = Q_{C, D, \hat{U}} \).

**Proof.** Let \( b := (C, D, A, e) \in B \) be the node in the call tree associated to the call to \( \text{Explore}(C, D, A) \). 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 \( en(C) = \emptyset \), so \( \hat{U} \subseteq Q_{C, D, \hat{U}} \). By hypothesis \( Q_{C, D, \hat{U}} \subseteq \hat{U} \) also holds, so \( \hat{U} = Q_{C, D, \hat{U}} \). Now, the call to \( \text{Extend}(C) \) adds to \( U \) only events from \( cex(C) \). So at line 4, clearly \( \hat{U} = Q_{C, D, \hat{U}} \).

**Step case.** Let \( U_1 := \hat{U} \) be the value of set \( U \) immediately before the function \( \text{Explore}(C, D, A) \) is called. Let \( U_2 \) be the value immediately before algorithm 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 \( \text{Explore}(C, D, A) \) 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 \( \tilde{e} \in [e_1] \) for some \( e_1 \in U_2 \) such that there is some \( e_2 \in C' \cup D \) with \( e_1 \neq e_2 \). Since \( U_2 \) 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 \( \tilde{e} \in Q_{C', D, U_2} \) (essentially 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 \( \text{Alt}(\cdot) \) 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 $D' := D \cup \{e\}$. 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 $\text{Remove}(e,C,D)$. 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:

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

(17)

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

To prove (17), first observe that the events removed from $U$ by $\text{Remove}(e,C,D)$, called $R$ above, are exactly

$$R := \left( \{e\} \cup \bigcup_{e' \in \#\_4(e)} [e'] \right) \setminus Q_{C,D,U_4}. \quad (18)$$

This is immediate from the definition of $\text{Remove}(\cdot)$. Now we prove two statements, (19) and (20), that imply the validity of (17). We start stating the first:

$$Q_{C,D\cup\{e\},U_4} \setminus R = Q_{C,D,U_4}. \quad (19)$$

This equality intuitively says that (left-hand side) executing $\text{Remove}(e,C,D)$ 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 in conflict with some event in $C \cup D$. For the shake of clarity, unfolding the definitions in (19) yields the following equivalent equality:

$$\left( C \cup D \cup \{e\} \cup \bigcup_{e' \in \#\_4(e)} [e'] \right) \setminus \left( \left\{ e \right\} \cup \bigcup_{e' \in \#\_4(e)} [e'] \right) \cap Q_{C,D,U_4} = Q_{C,D,U_4}$$

We now prove (19). 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 $\tilde{e} \not\in R$. Now, from the definition (18) of $R$ we get that $\tilde{e} \in Q_{C,D,U_4}$. Lastly, if $\tilde{e} \in C \cup D \cup \{e\}$, then there is some event $e' \in C \cup D \cup \{e\}$ and some event $e'' \in U_4$ such that $e' \#^{\tilde{e}} e''$ and $\tilde{e} < e''$. If $e' \in C \cup D$, then by definition $\tilde{e} \in Q_{C,D,U_4}$. 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 $\tilde{e} \not\in R$, then $\tilde{e}$ is either in $Q_{C,D,U_4}$, as we wanted to show, or $\tilde{e}$ is not in $\{e\} \cup \bigcup_{e' \in \#\_4(e)} [\tilde{e}]$. 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 \( \tilde{e}'' \in U_4 \) such that \( e' \preceq \tilde{e}'' \) and \( \tilde{e} \preceq \tilde{e}'' \). Then by definition \( \tilde{e} \) is in the left-hand side. This completes the proof of (19).

The second statement necessary to prove (17) is the following:

\[
Q_{C,D,U_4} = Q_{C,D,U_5}
\] (20)

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 \( e_2 \in \#_{U_4}(e_1) \) 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 (18), 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 \( e_2 \in \#_{U_5}(e_1) \) 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: Eager Exploration Algorithm

- Algo 2 also defines a call graph, nodes have the same shape

Lemma 15. If \( \{B, \triangleright\} \) is the graph generated by one execution of algorithm 2, then it is also the graph generated by some execution of algorithm 1.

Proof. Both algorithm 1 and algorithm 2 are non-deterministic. We need to show that for every choice made by algorithm 2 it is possible to resolve non-determinism in algorithm 1 so as to imitate the choice in algorithm 2. Each algorithm exhibits non-determinism when (a) choosing the next event to add to the stack (as in general \( A \cap en(C) \) is not a singleton) as well as when (b) choosing some alternative \( J \) to \( D \cup \{e\} \) after \( C \). The way both algorithms differ when choosing an alternative is the only relevant difference between them for this proof.

Consider the in-order \( \preceq \subseteq B \times B \) defined as in the proof of Lemma 12. That is, the only order in \( B \) order that sorts, for every \( b \in B \), first all nodes reachable from \( b \)'s left child (if there is any), then \( b \), then all nodes reachable from \( b \)'s right child (if there is any). We show the following two claims:

1. If algorithm 2 \( b_1 \triangleright_1 b_2 \triangleright_1 \ldots \triangleright_1 b_n \) is the leftmost branch of \( \{B, \triangleright\} \), then algorithm 1 can explore the same nodes in the same order.
2. Given a node \( b := (C,D,e) \in B \), assume that algorithm 1 and algorithm 2 have explored all nodes in \( \{b' \in B : b' < b\} \) exactly in the same order. Then node \( b \) has a right-hand side child in \( B \) iff algorithm 1 finds an alternative to \( D \cup \{e\} \) after \( C \). Furthermore, if \( b \) has a right-hand side child \( b' \), then algorithm 1 can explore the leftmost branch rooted at \( b' \) in exactly the same order as algorithm 2 will explore that branch.
These two facts are sufficient to prove the lemma, as we argue now. By the first fact, when algorithm 2 decides which event it fires (and in which order) in order to explore the first maximal configuration, algorithm 1 can mimic all moves. Then algorithm 2 backtracks to some node $b$. By the second claim both algorithms agree on whether $b$ has a right-hand side child $b'$. If it has such $b'$, then both can also explore the same configurations (in the same order) until they arrive to the leftmost maximal configuration. From there we can iterate exactly the same argument one more time to show how algorithm 1 can be ‘forced’ to explore configurations in the same order as algorithm 2 does. Since every time we iterate we discover new nodes, and the call tree of algorithm 1 is finite (Corollary 2), then necessarily this argument terminates.

We now prove the previous two claims:

1. **Claim 1.** Let $b = \{C, \ldots, \cdot\} \in B$ be the leftmost leaf in $B$, explored by algorithm 2. Trivially algorithm 1 can also explore this maximal configuration, and can append events to $C$ in the same order as algorithm 2 did.

2. **Claim 2.** Let $b := (C, D, A, e) \in B$ be node in the call tree of algorithm 2. Assume both algorithms have explored all nodes in $\{b' \in B : b' \prec b\}$ exactly in the same order.

   - If $b \succ_r b'$, then algorithm 1 finds a right child. Assume that algorithm 2 found a right child $b'$. Let $\hat{C}$ be the leftmost maximal configuration in the subtree rooted at $b'$. By Lemma 12, since $C \subseteq \hat{C}$, $D \cap \hat{C} = \emptyset$, and $e \notin \hat{C}$, then algorithm 1 necessarily finds a right child as well. Importantly, the proof of Lemma 12 constructs some alternative $J \in \text{Alt}(C, D \cup \{e\})$ such that $J \subseteq \hat{C}$. This fact will be used later.

   - If algorithm 1 finds a right child, then $b \succ_r b'$. See Lemma 16.

   - Algorithm 1 can be forced to explore the leftmost branch rooted at $b'$ in exactly the same order as algorithm 2. Let $\hat{C}$ be the leftmost maximal configuration explored by algorithm 2 in the subtree rooted at $b'$. We showed in Lemma 12 that algorithm 1 can pick from $\text{Alt}(C, D \cup \{e\})$ some alternative $J$ such that $J \subseteq \hat{C}$. We make algorithm 1 take this alternative. Observe that $J$ will be in general different than the alternative $\tilde{J} \in \text{Alt}_2(C, D \cup \{e\})$ which algorithm 2 took. But in both cases $J \subseteq \hat{C}$ and $\tilde{J} \subseteq \hat{C}$. It is now trivial to force algorithm 1 explore the events of $J \setminus C$ and later those of $\hat{C} \setminus (C \cup J)$ to imitate the exploration made by algorithm 2.

**Lemma 16.** **[César: I want to shoot myself!!]**

**Proof.** This lemma.

**Theorem 6 (Termination, optimality, completeness).** **Algorithm 2 always terminates and calls exactly once function Explore2($C, \ldots, \cdot$) for every maximal configuration $C$.**

**Proof.** By Lemma 15, whenever algorithm 2 executes, it generates a call tree isomorphic to one that algorithm 1 can generate. Since all call trees generated
Lemma 17. Let $b := (C, D, A, e) \in B$ and $b' \in B$ be nodes in the call graph of algorithm 2 such that $b \triangleright^* b'$ and $b'$ is a leaf node. Assume that $\text{Explore}(\cdot)$ adds configuration $J$ to $\text{A}(e)$ when visiting $b'$. Then, when $\text{Explore}(C, D, A)$ reaches line 14 it holds that $J \subseteq U$ (i.e., no event from $J$ has been removed).

Proof. Function $\text{Explore}(\cdot)$ recursively visits all nodes in the subtree rooted at $b$, exploring leaf nodes from left to right. At some point it discovers a leaf node $b'$ and adds $J$ to $\text{A}(e)$. Let $U_1$ be the set of events contained in the formal variable $U$ of algorithm 2 just after $J$ is added to $\text{A}(e)$. We need to show that all events in $J$ remain in $U$ until $\text{Explore}(\cdot)$ eventually backtracks to $b$ and reaches the last line of algorithm 2.

Events are removed from $U$ only when $\text{Explore}(\cdot)$ backtracks. Let $\tilde{b} := \tilde{(C, \tilde{D}, \tilde{A}, \tilde{e})} \in B$ be any node from which $\text{Explore}(\cdot)$ backtracks after having explored $b'$. Such $b'$ satisfies $b \triangleright^* b \triangleright^* b'$. Since $b \triangleright^* b$, clearly either $e \in \tilde{C}$ or $e \in \tilde{D}$. So when $\text{Explore}(\tilde{C}, \tilde{D}, \tilde{A})$ calls $\text{Remove2}(\tilde{e})$, clearly $J \subseteq Q'_{\tilde{C}, \tilde{D}}$. Since $\text{Remove2}(\tilde{e})$ never removes any event from $Q'_{\tilde{C}, \tilde{D}}$ from $U$, all events in $J$ remain intact in $U$ after $\text{Remove2}(\tilde{e})$ returns.

D Proofs: Improvements

- exploration with cutoffs defines a call graph, like before - many properties hold like before, point to specific ones, e.g., $D \subseteq ex(C)$
- prove termination and optimality, possibly handwaving
- the algorithm therefore explores a set of maximal configurations $C_1, \ldots, C_n$
- define $P_1 := \langle \mathcal{E}_1, <, \# \rangle$ the red prefix (all events for which, whenever asked, the algorithm never ever says the event is a cutoff) - define $P_2 := \langle \mathcal{E}_2, <, \# \rangle$ the blue prefix (union of all explored maximal)
- Let $P_M$ the prefix constructed with the following cutoff criterion: an event $e$ is cutoff iff there exists $u \in U_M$ another event $e'$ such that FIXME. - Clearly every event in $P_M$ is a red event, as regardless of the actual contents of $U$ and $G$, the cutoff predicate will always answer NO. - So we have $P_M \subseteq P_1$.
- A call tree is complete iff for every $s \in \text{reach}(M)$ reachable by $M$, the call tree contains a node $b := \langle C, \ast, \ldots \rangle$ such that some configuration $C' \in C$ is such that $\text{state}(C') = s$.
- So, in order to prove that the call tree generated by the lazy algorithm retrofitted with cutoff detection is complete, it suffices to show that every red configuration from $P_1$ is contained in some node explored the algorithm. We achieve this with Lemma 18 and Lemma 19.
- Upgrading the cutoff detection algorithm to use total adequate orders is straightforward, and this proof method will guarantee (without modification!) that the explored call tree is complete.
Lemma 18. Let \( b := (C, D, A, e) \in B \) 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 \( \hat{e} \in D \) there is some \( e' \in \hat{C} \) such that \( \hat{e} \#^i e' \).

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 \(- (e \#^i \hat{e}) \) 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 12, the main lemma behind the proof of Theorem 5 (completeness). Consequently the proof is also similar. The proof is by induction on \( b \) using the same total order \( \ll \in B \times B \) that we employed for Lemma 12.

Base case. Node \( b \) is the least element in \( B \) w.r.t. \( \ll \). 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 \( \tilde{b} \ll b \). 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 \( J \in \text{Alt}(C, D \cup \{e\}) \) whenever the algorithm asks for the existence of one.

We define the set

\[
F := \{e_1, \ldots, e_n\} := D \cup \{e\}.
\]

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

\[
J := \{\{e'_1, \ldots, e'_n\}\}
\]

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 \( \mathcal{P}_1 \). 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 \( J \in \text{Alt}(C, D \cup \{e\}) \) when function \( \text{Alt}(\cdot) \) is called just before line 11 during the execution of \( \text{Explore}(C, D, A) \). Let \( \hat{U} \) be the set of events contained in the variable \( U \) exactly when \( \text{Alt}(\cdot) \) 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 \( \text{Alt}(\cdot) \) is called.
We reason about the call stack when the algorithm is situated at \( b = (C, D, A, e) \).

For \( i \in \{1, \ldots, n\} \) let \( b_i := (C_i, D_i, A_i, e_i) \in B \) be the node in the call graph associated to event \( e_i \in E \). 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

\[
b_0 \triangleright^* b_1 \triangleright^* b_2 \triangleright^* \ldots \triangleright^* b_n,
\]

where \( b_n = b \) and \( e_n = e \). 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. Also, remark that by construction we have \( \{e_1, \ldots, e_{i-1}\} = D_i \) for every \( i \in \{2, \ldots, n\} \).

We need to show that \( e_i' \in \hat{U} \), for \( i \in \{1, \ldots, n\} \). We regard two cases. Consider the set \( D_i = \{e_1, \ldots, e_{i-1}\} \). Only two things are possible: either there exists some \( j \in \{1, \ldots, i-1\} \) such that

\[
\#(e_j) \cap \hat{C} \subseteq \#(e_i) \cap \hat{C}
\]

(21)

holds, or for all \( j \in \{1, \ldots, i-1\} \) the above statement is false.

- **Case 1**: for all \( j \in \{1, \ldots, i-1\} \) we have that (21) 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 \( \hat{e} \in D_i \) there is some \( e' \in X_1 \) such that \( \hat{e} \not\equiv^i \ e' \). This holds by construction. For any \( \hat{e} \in D_i = \{e_1, \ldots, e_{i-1}\} \) we know that some event in \( \#(\hat{e}) \cap \hat{C} \) is not in \( \#(e_i) \cap \hat{C} \), so it is necessarily in \( X_1 \).

Consider the left child \( b_i' := (C_i \cup \{e_i\}, D_{i'}, \cdot, \cdot) \) of \( b_i \). Every node \( \hat{b} \) in the subtree rooted at \( b_i' \) (i.e., \( b_i' \triangleright^* \hat{b} \)) is such that \( \hat{b} < b_i < b \). The induction hypothesis thus applies to \( \hat{b} \). By the previous facts, Lemma 19 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 \( e_i' \in \text{ex}(C') \), 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 \( e_i' \in \hat{U} \), what we wanted to prove.

- **Case 2**: there is some \( j \in \{1, \ldots, i-1\} \) such that (21) holds. Let \( m \) be the minimum such integer. Consider the set \( X_2 \) defined as

\[
X_2 := \hat{C} \setminus \#(e_i) \cup [e_m']
\]

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 \([e'_m]\) we do no add any conflict, as there is no conflict between \(e_m\) and any event of \([e'_m]\).

• **Fact 4:** for any \(\tilde{e} \in D_m\) there is some \(e' \in X_2\) such that \(\tilde{e} \neq e'\). This holds by construction, due to the minimality of \(m\). For any \(\tilde{e} \in D_m = \{e_1, \ldots, e_{i-m}\}\) we know that (21) 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., \(b'_m \triangleright^* \hat{b}\)). By the previous facts, **Lemma 19** 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 (21), 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 \(e_i \neq \tilde{e}\). 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 choose 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 \(J \in Attr(C, D \cup \{e\})\) 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 19.** Let \(b := \langle C, D, \cdot, \cdot, e \rangle \in B\) 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 \(\tilde{e} \neq e'\);
- **Lemma 18** 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 \(b \triangleright^* b'\) and \(\hat{C} \subseteq C'\).

**Proof.** Assume that **Lemma 18** holds on any node \(b'' \in B\) such that \(b \triangleright^* b''\), i.e., all nodes in the subtree rooted at \(b\). By hypothesis we can apply **Lemma 18** 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 18 applies to $b_1$ as well.

If $e$ is in conflict with some event in $\hat{C}$, then by Lemma 18 we know that $b_1$ 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 18 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., $\text{en}(C_n)$ 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 7 (Completeness).** Algorithm 1 updated with the cutoff mechanism described above is complete.

**Proof.** Let $P_M$ be an unfolding prefix constructed with the traditional breath-first unfolding algorithm, using McMillan’s cutoff criterion: an event $e$ is a $m$-cutoff if there is another event $e'$ in $U_M$ such that $\text{state}([e]) = \text{state}([e'])$ and $|[e']| < |[e]|$. By construction all events in $P_M$ are in $P_1$.

Let $\langle B, \triangleright \rangle$ be the call tree associated with one execution of algorithm 1 retrofitted with the cutoff mechanism. Let $s \in \text{reach}(M)$ be an arbitrary state of the system. Due to the properties of $P_M$ [César: cite] there is a configuration $\hat{C}$ in $P_M$ such that $\text{state}(\hat{C}) = s$. Such configuration is in $P_1$.

Now, Lemma 19 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.