Unfolding-based Dynamic Partial Order Reduction for LTL Model Checking

Moussaoui Remil, Naïm. Jéron, Thierry. Quinson, Martin.

Abstract

Unfolding-based Dynamic Partial Order Reduction (UDPOR) is a recent technique mixing Dynamic Partial Order Reduction (DPOR) with concepts of concurrency such as unfoldings to efficiently migitate state space explosion in model-checking of concurrent programs. To achieve this, UDPOR used the notion of cutoff that stops the exploration of an interleaving when a known event with some conditions is rediscovered. In other words, loops are cut in the exploration of programs with UDPOR. Model checking of LTL being based on the detection of loops is then infeasible with the current UDPOR algorithm. In this paper, we propose a refined cut-off notion to detect loops and so a refined version of UDPOR that allow LTL-X model checking.

1 Introduction

Model-checking is a set of techniques allowing verification automatically and effectively of some properties on distributed systems. The principle is usually to explore all possible behaviors (states and transitions) of the system model. However, state spaces increase exponentially with the number of concurrent processes. Unfoldings and Partial order reduction (POR) are two candidate alternative techniques born in the 90's to mitigate this state space explosion and scale to large applications.

POR comprises a set of explorations techniques (see [6]), sharing the idea, that to detect deadlocks (and by extension, for checking safety properties) it is sufficient to cover each Mazurkiewicz trace, i.e. a class of interleavings equivalent by commutation of consecutive independent actions. This state-space reduction is obtained by choosing at each state, based on the independence of actions, only a subset of actions to explore (ample, stubborn, or persistent sets, depending on the method), or to avoid (sleep set). Dynamic Partial Order Reduction (DPOR) [5], was later introduced to combat state space explosion for stateless model-checking of software. In this context, while POR relies on a statically defined and imprecise independence relation, DPOR may be much more efficient by dynamically collecting it at run-time. Nevertheless, redundant explorations, named sleep-set blocked (SSB), may still exist that would lead to an already

visited interleaving, and detected by using sleep sets.

Unfolding (see [3]) is a concept of concurrency theory providing a representation of the behaviors of a model in the form of an event structure aggregating causal dependencies or concurrency between events (occurrence of actions), and conflicts that indicate choices in the evolution of the program.
In general, the unfolding of a program-model creates an infinite structure due to the presence of loops in the model. To handle this McMillan [7] has introduced the notion of cutoff. While unfolding the model we can stop the expansion of the Event Structure when we rediscover a known event with a shorter history. It allows us to cut the infinite discovery of a loop. Thus, the unfolding with cut-off is finite.
This (finite) representation may be exponentially more compact than an interleaving semantics, while still allowing to verify some properties such as safety. However the usual notion of cutoff makes liveness properties and LTL in general not verifiable.

In the last few years, two research directions were investigated to improve DPOR. The first one tries to refine the independence relations: the more precise, the fewer Mazurkiewicz traces exist, thus the more efficient could be DPOR.
For example, [2] proposes to consider conditional independence relations where commutations are specified by constraints, while in [4] independence is built lazily, conditionally to sfuture actions called observers. The other direction proposes alternatives to persistent sets, to minimize the number of explored interleavings.
Optimality is obtained when exactly one interleaving per Mazurkievicz trace is built. In [9] the authors propose unfolding-based DPOR (UDPOR), an optimal DPOR method combining the strengths of PORs and unfolding with the notion of alternatives. Despite being optimal UDPOR makes liveness properties not verifiable by inheritance of the unfolding semantics.

In [4] the author have already studied the LTL verification problem for Petri nets unfolding. They solved it by refining their cut-off to explore more events and being able to detect loops. In our context we will take the same approach.

We propose in this paper a version of UDPOR to make liveness and more generally LTL-X properties verifiable.
The paper is organized as follows. Section 2 recalls notions of unfolding and partial order reductions along with their combination in UDPOR and the automata approach of LTL model checking. Section 3 presents the new cut-off. Section 4 presents our adapted UDPOR .

2 Preliminaries

2.1 Model Checking of LTL: the Automata Approach

The behaviors of a distributed program can be described in an interleaving semantics by a labeled transition system, or in a true concurrency semantics by an event structure.

Définition .1. A labelled transition system (LTS) is a tuple $\mathcal{T} = \langle S, s_0, \Sigma, \rightarrow \rangle$ where S is a set of states, $s_0 \in S$ the initial state, Σ is the alphabet of actions, and $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation.

We note $s \xrightarrow{a} s'$ when $(s, a, s') \in \rightarrow$ and extend the notation to execution sequences: $s \xrightarrow{a_1.a_2...a_n}$ if $\exists s_0 = s, s1, ...s_n = s'$ with $s_{i-1} \xrightarrow{a}_i s_i$ for $i \in [1, n]$.
For a state s in S, we denote $enabled(s) = \{a \in \Sigma : \exists s' \in S, s \xrightarrow{a} s'\}$ the set of actions enabled at s.

The goal of model checking is to verify properties on distributed system's model (here a LTS). In our context, we would like these properties to express logical time. To fulfill this purpose, we will use Linear Time Logic (LTL) and more precisely the LTL-X fragment. For convenience reasons, we use an action-based temporal logic.

Définition .2. Given a finite set of actions AP, the abstract syntax of LTL-X is defined as followed:
$\phi := a \in AP | \neg\phi | \phi_1 \wedge \phi_2 | \phi_1 \mathcal{U} \phi_2$

The semantic of LTL-X is a set of infinite words over the 2^{AP} alphabet. However because two actions cannot happen simultaneously the alphabet can be restrict to set of singleton in 2^{AP} which is isomorphic to AP. Furthermore, for every LTL-X formula ϕ we can associate its $\omega - language$: $\mathcal{L}(\phi)$.

Now we can define the model checking problem. Given a LTS $\mathcal{T}$ over AP and LTL formula ϕ over AP we want to know if $P \vDash \phi$ hold. To resolve this problem we use the classical approach:

- Construct the Büchi Automaton $A_{\neg\phi}$ that accept all words satisfying $\neg\phi$,
- Construct the product $\mathcal{T} \otimes A_{\neg\phi}$,
- search a path in $\mathcal{T} \otimes A_{\neg\phi}$ that meets the acceptance condition of $A_{\neg\phi}$,
- If we find such a path then $\mathcal{T} \nvDash \phi$
  Else $\mathcal{T} \vDash \phi$.

Let's introduce needed definitions.

Définition .3. A Non-deterministrict Büchi Automata (NBA) is tuple $A = (Q, \Sigma, \delta, I, F)$ that consist of:

- Q is a finite set which elements are the states of A,
- Σ is the alphabet of A,
- $\Delta \in Q \times \Sigma \times \Sigma$ is the transition relation of A,
- I is the set of initial state,
- $F \subseteq Q$ is the set of terminal state. A accepts exactly those infinite runs in which at least one elements of F occurs **infinitely often**.

Définition .4. The product of a labelled transition system $\mathcal{T} = \langle S, s_0, \Sigma, \rightarrow \rangle$ and an NBA $A = (Q, \Sigma, \Delta, I, F)$ is a tuple $\mathcal{T} \otimes A = \langle S \times Q, \Sigma, \rightarrow' \rangle$ where $\rightarrow'$ is the minimal transition relation defined by the rule : $[]s$ -⟩s

Hence, the product $\mathcal{T} \otimes A_{\neg\phi}$ is a LTS containing a transition $\langle s, q \rangle \xrightarrow{\alpha'}{}' \langle s', q' \rangle$ for every $q \xrightarrow{\alpha} q'$ in $A_{\neg\phi}$ and for every s such that $\exists s' \in T, s' \in \Delta(s, \alpha)$.

For the LTL model checking we use a subset $V \cup \{\tau\}$ with $V \subseteq \Sigma$ to create the product $\mathcal{T} \otimes A_{\neg\phi}$ instead of the entire set Σ. V is the set of visible actions, in practice it is the set of action appearing in ϕ. The action τ is used to represent the invisible actions. This syncrhonisation is more efficient (in the size of the product). However, it creates a new type of traces with invisible actions. It leads us to split the infinite traces of $\mathcal{T} \otimes A_{\neg\phi}$ into two sets. In one hand, the illegal infinite-trace whose execution sequences are as follows: $\langle s_1, q_1 \rangle \xrightarrow{a_1} \langle s_2, q_2 \rangle \xrightarrow{a_2} \langle s_3, q_3 \rangle ...$ with infinitely many $\langle s_i, q_i \rangle$ such that q_i is a terminal state of $A_{\neg\phi}$. In the other hand, the illegal livelock whose executions sequences are as follow: $\langle s_1, q_1 \rangle \xrightarrow{a_1} \langle s_2, q_2 \rangle \xrightarrow{a_2} \langle s_3, q_3 \rangle ... \langle s_{n-1}, q_{n-1} \rangle \xrightarrow{a_{n-1}} \langle s_n, q_n \rangle ..$ with $a_i = \tau$ for every $i \rangle n - 1$.

This synchronization leads to the following theorem which replaces the third point of the LTL model checking procedure.

Théorème .1. *Let $\mathcal{T}$ be a labelled transition system and ϕ a LTL-X formula. $\mathcal{T} \vDash \phi$ if and only if the product $\mathcal{T} \otimes A_{\neg\phi}$ has no illegal infinite trace and no illegal livelock.*

Now the third point of the LTL model checking procedure is :

- search a path in $\mathcal{T} \otimes A_{\neg\phi}$ that is an illegal infinite-trace or an illegal livelock.

The major drawback of the current procedure is the exploration/construction of the set of execution sequences in $\mathcal{T} \otimes A_{\neg\phi}$. To do so a naive approach is to explore all possible executions of $\mathcal{T} \otimes A_{\neg\phi}$. However, as we mentioned in the introduction the size of the state spaces increase exponentially with the number of concurrent processes i.e. the number of executions in $\mathcal{T} \otimes A_{\neg\phi}$ increases exponentially. To reduce this space of executions we will use Unfolding-based Dynamic Partial Order Reduction (UDPOR) on $\mathcal{T} \otimes A_{\neg\phi}$.

2.2 UDPOR

An LTS equipped with an independence relation can be unfolded into an event structure [9]. This is the main step for UDPOR.

Independence is a key notion in both POR techniques and unfoldings linked to the possibility to commute actions:

Définition .5. Two actions a_1, a_2 of a LTS $\mathcal{T} = \langle S, s_0, \Sigma, \rightarrow \rangle$ commute in a state s if they satisfy two conditions:

- executing one action does not enable nor disable the other one :
  $a_1 \in enabled(s) \wedge s \xrightarrow{a_1} s' \Rightarrow (a_2 \in enabled(s) \Leftrightarrow a_2 \in enabled(s'))$ (1)
- their execution order does not change the overall result:
  $a_1, a_2 \in enabled(s) \Rightarrow (s` \xrightarrow{a_1.a_2} s' \wedge s` \xrightarrow{a_2.a_1} s'' \Rightarrow s' = s'')$ (2)

A *relation* $I \subseteq \Sigma \times \Sigma$ is a valid independence relation if it under-approximates commutation; i.e for all $a_1, a_2, I(a_1, a_2)$ implies that a_1 and a_2 commute in all states. Conversely a_1 and a_2 are dependent and we note $D(a_1, a_2)$ when $\neg(I(a_1, a_2))$
A *Mazurkiewicz trace* is an equivalence class of executions (or interleavings) of an LTS $\mathcal{T}$ obtained by commuting adjacent independent actions. By the second item of Definition 2, all these interleavings reach a unique state. The principle of all DPOR approaches is precisely to reduce the state space exploration while covering at least one execution per Mazurkiewicz trace. If a deadlock exists, a Mazurkiewicz trace leads to it and will be discovered. More generally, safety properties are preserved. The UDPOR technique that we consider also uses concurrency notions. A classical model of true concurrency is prime event structures:

Définition .6. Given an alphabet of actions Σ, a $\Sigma - prime$ event structure ($\Sigma - PES$) is a tuple $\mathcal{E} = \langle E, <, \#, \lambda \rangle$ where E is a set of events, $\langle$ is a partial order relation on E, called the causality relation, $\lambda : E \rightarrow \Sigma$ is a function labelling each event e with action(e), $\#$ is an irreflexive and symmetric relation called the conflict relation such that, the set of causal predecessors or history of any event e, $\lceil e \rceil = \{e' \in E : e' < e\}$ is finite and conflicts are inherited by causality: $\forall\ e, e', e" \in E, e\#e' \wedge e' \langle e'' \Rightarrow e\#e''$

Intuitively, $e < e'$ means that e must happen before e', and $e\#e'$ that those two events cannot belong to the same execution. Two distinct events that are neither causally ordered nor in conflict are said *concurrent*. The set $[e] = \lceil e \rceil \cup \{e\}$ is called the local configuration of e.
An event e can be characterized by a pair $\langle \lambda(e), H \rangle$ where $\lambda(e)$ is its action, and $H = \lceil e \rceil$ its history. We note $conf(E)$ the set of configurations of $\mathcal{E}$, where a configuration is a set of events $C \subseteq E$ that is both causally closed ($e \in C \Rightarrow \lceil e \rceil \subseteq C$) and conflict free ($e, e' \in C \Rightarrow \neg(e\#e')$). A configuration C is characterized by its causally maximal events $maxEvents(C) = \{e \in C : \nexists e' \in C, e < e'\}$, since it is exactly the union of local configurations of theses events :

$C = \bigcup_{e \in maxEvents(C)} [e]$; conversely a conflict free set K of incomparable events for $\langle$ defines a configuration $config(K)$ and $C = config(maxEvents(C))$.
A configuration C, together with the causal independence relation defines a *Mazurkiewicz trace*: all interleavings are obtained by causally ordering all dependent events in the configuration but commuting concurrent ones. The state of a configuration C denoted by $state(C)$ is the state in $\mathcal{T}$ reached by any of these executions, and it is unique as discussed above. We write $enab(C) = enabled(state(C)) \in \Sigma$ for the set of actions enabled at $state(C)$, while $action(C)$ denotes the set of actions labelling in C.
The set of *extensions* of C is $ex(C) = \{e \in E \setminus C : \lceil e \rceil \subseteq C\}$, *i.e* the set of events not in C but whose causal predecessors are all in C. When appending an extension to C, only resulting conflict-free sets of events are indeed configurations.
These extensions constitute the set of enabled events $en(C) = \{e \in ex(C) : \nexists e' \in C, e \# e'\}$ while the other ones are conflicting extensions $cex(C) := ex(C) \backslash en(C)$.

Parametric Unfolding Semantics. Given an LTS $\mathcal{T}$ and an independence relation I, one can build a prime event structure E such that each linearization of a maximal (for inclusion) configuration represents an execution in $\mathcal{T}$, and conversely, to each Mazurkiewicz trace in $\mathcal{T}$ corresponds a configuration in E [10].

Définition .7. The unfolding of an LTS $\mathcal{T}$ under an independence relation I is the $\Sigma - PES$ $\mathcal{E} = \langle E, <, \#, \lambda \rangle$ incrementally constructed from the initial $\Sigma - PES$ $\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ by the following rules until no new event can be created:

- for any configuration $C \in conf(E)$, any action $a \in enabled(state(C))$, if for any $e_0 \in maxEvents(C), I(a, \lambda(e_0))$, add a new event $e = \langle a, C \rangle$ to E;
- for any such new event $e = \langle a, C \rangle$, update $<, \#$ and λ as follows: $\lambda(e) := a$ and for every $e_0 \in E \setminus e$, consider three cases:
  1. if $e_0 \in C$ then $e_0 < e$
  2. if $e_0 \notin C$ and $(a, \lambda(e_0))$, then $e \# e_0$
  3. otherwise, i.e. if $e_0 \notin C$ and $I(a, \lambda(e_0))$, then e and e_0 are concurrent.

This definition of unfolding can lead to an infinite prime event structure. For example, consider an LTS with a loop. In order to obtain a finite unfolding, we introduce the notion of cutoff.

Définition .8. Let $\mathcal{T}$ be a LTS and $\mathcal{E} = \langle E, <, \#, \lambda \rangle$ be a finite prefix of $\mathcal{T}$'s unfolding. An event e is a cutoff if and only if there is an event $e' \in E$ such that $state(e) = state(e')$ and $|[e']| < |[e]|$. We will call e' the representative of e.

The intuition of a cutoff is simple. While extending the unfolding we declare a known event as a cutoff if we already know an event in the unfolding whose local configuration is shorter and has the same state. We suppose a boolean

function Cutoff(e) returning true if e is a cutoff and false if it's not. Now we can define a finite unfolding.
As our goal is to handle the model checking problem we will propose a definition for a product $\mathcal{T} \otimes A = \langle S \times Q, \Sigma, \rightarrow' \rangle$. To do it, we need to update the definition of $enabled(\_)$. For any state $\langle s, q \rangle \in S \times Q$ we redefined $enabled(\langle s, q \rangle) = \{\langle a, q' \rangle : \exists \langle s', q' \rangle \in S \times Q, \langle s, q \rangle \xrightarrow{a} \langle s', q' \rangle\}$.

Définition .9. Given a LTS $\mathcal{T}$ and a NBA A. The unfolding of the product $\mathcal{T} \otimes A$ under an independence relation I is the $\Sigma - PES$ $\mathcal{E} = \langle E, <, \#, \lambda \rangle$ incrementally constructed from the initial $\Sigma - PES$ $\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ by the following rules until no new event can be created:

- for any configuration $C \in conf(E)$, any pair $\langle a, q \rangle \in enabled(state(C))$, if for any $e_0 \in maxEvents(C), (a, \lambda_1(e_0))$, if $\neg\ Cutoff(\langle a, q, C \rangle)$ then add a new event $e = \langle a, q, C \rangle$ to E;
- for any such new event $e = \langle a, q, C \rangle$, update $<, \#$ and λ as follows: $\lambda(e) := a$ and for every $e_0 \in E \setminus e$, consider three cases:
  1. if $e_0 \in C$ then $e_0 < e$
  2. if $e_0 \notin C$ and $(a, \lambda(e_0))$, then $e \# e_0$
  3. otherwise, i.e. if $e_0 \notin C$ and $I(a, \lambda(e_0))$, then e and e_0 are concurrent.

3 Detecting illegal infinite trace

In this section we consider a LTL-X formula ϕ, its corresponding Büchi Automata A_ϕ and LTS P.

In [9], the authors use the notion of cutoff to obtain a finite unfolding. Cutoff stops the exploration of loops thus the exploration algorithm presented in [9] limits the model checking to safety properties. To handle the model checking of LTL-X we have to work a little bit. Theorem 1 splits the procedure into two parts: we have to detect illegal infinite traces and illegal livelocks in $\mathcal{T} \otimes A_{\neg\phi}$. In this section we will answer the first problem.

The detection of illegal infinite traces consists in detecting loops in the unfolding of $\mathcal{T} \otimes A_{\neg\phi}$ in which an event $\langle a, H, q \rangle$ where q is a terminal state of $A_{\neg\phi}$ appears.

First, we notice that we detect a loop if and only if we have a cut-off. Thus the problem of detecting a loop is the same as detecting a cut-off. Now we have to find out a way to detect if these loops represented by cut-off are illegal infinite traces or not.
To do so, we first notice that the unfolding of a transition system is a tree.

Hence, a loop can be represented by a cutoff in the same branch as its representative or in another branch.
Dealing with cutoff in the same branch as its representative is easy. The idea is whenever a cutoff $e = \langle a, H, q\rangle$ with representative $e' = \langle a', H, q'\rangle$ with $e < e'$ is detected. We check if in $H' \setminus H$ there is an event $\langle a_t, H_t, q_t\rangle$ with q' a terminal state of $A_{\neg\phi}$.
If yes, then we can exhibit an illegal infinite trace which is the linearization of the configuration $H' \setminus H$.

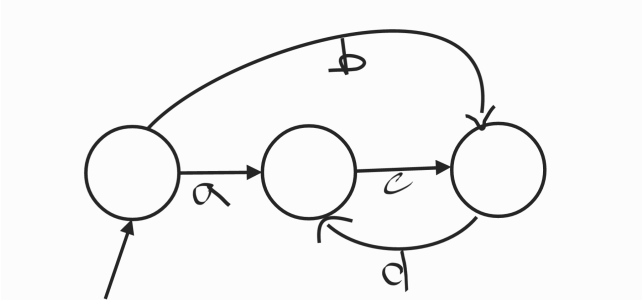
For the cutoff detected in another branch, this is not that simple. In [4], the problem of LTL model checking has already been handled with the unfolding of Petri nets. For that, they have extended their definition of the cut-off as follows:

Définition .10. Let $\mathcal{T}$ be a LTS and $\mathcal{E} = \langle E, <, \#, \lambda\rangle$ be a finite prefix of $\mathcal{T}$'s unfolding. An event e is a cutoff if and only if there is an event $e' \in E$ such that $state(e) = state(e')$, $|[e']| < |[e]|$ and:

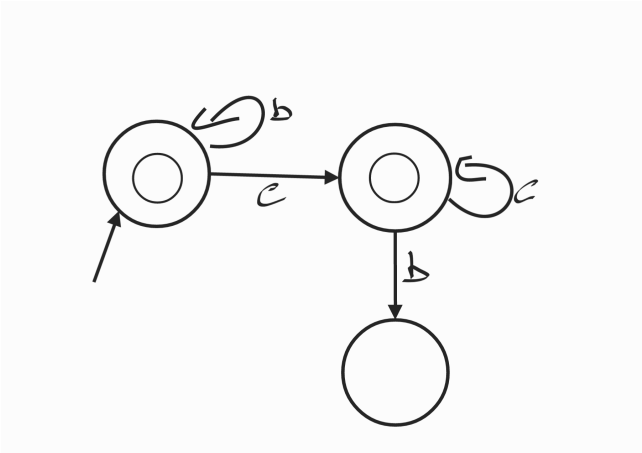
- $e < e'$ or
- $\neg(e < e')$, $[e'] \prec [e]$ and $\#_F(H') \geq \#_F(H)$, where $\#_F(H)$ is the number of events with a transition leading to terminal state.

Their definition of cut-off is still necessary and sufficient in our context. Let's show that this is a necessary condition with the following example.

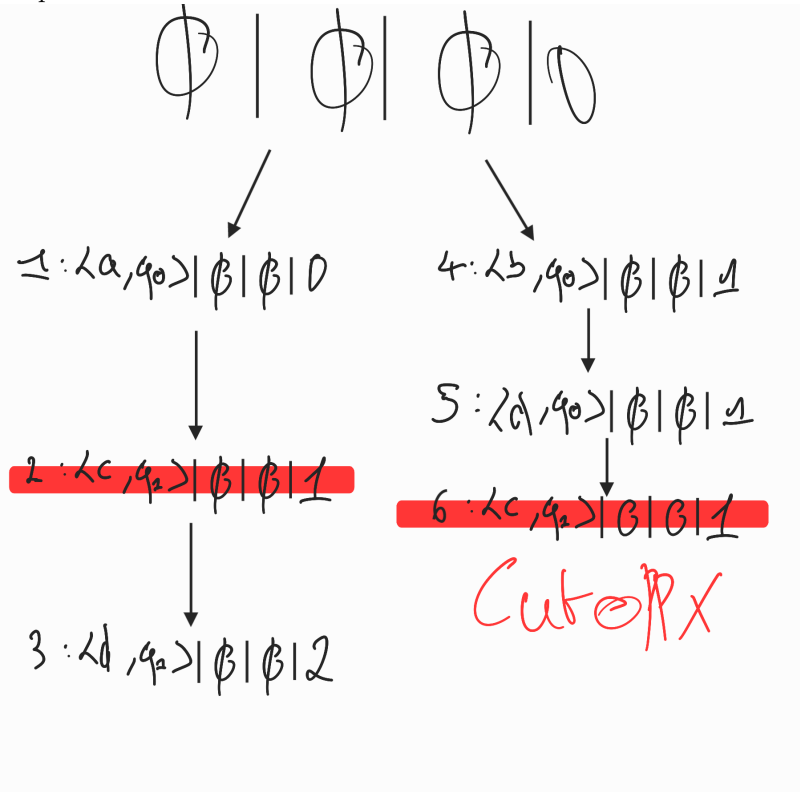
Here we consider the set of atomic propositions $AP = \{a, b, c, d\}$ with the visible actions $\{c, d\}$ and the following LTS "$\mathcal{T}$".



We want to verify the formula $\phi = \Diamond(c \wedge \Diamond b)$ which is not satisfied by the LTS, thus we must find a counterexample. Thus we create the Büchi Automaton $A_{\neg\phi}$ of $\neg\phi = \Box(\neg c \vee \Box\neg b)$

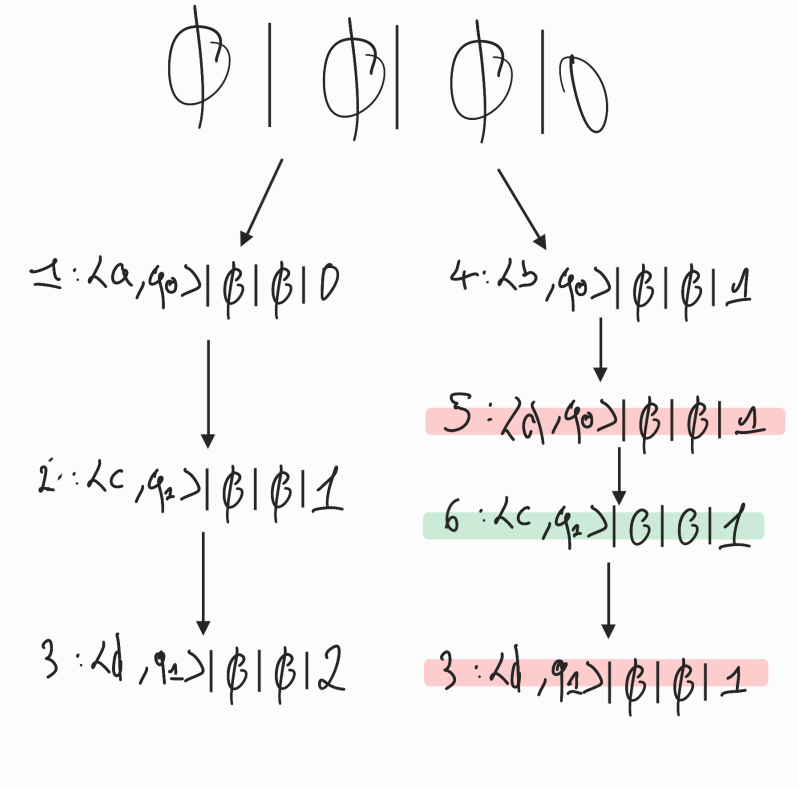


With the classical cutoff definition, the exploration algorithm of [9] will give us this exploration tree.



The representation of the nodes is like in [9] plus a counter that we will detail in the next section. In this counter-example, we see that the exploration is stopped when discovering event number 6 because we already know event 5 and $state(e_6) = state(e_5)|[e_5]| = 2\langle|e_6| = 3$. Thus we obtain a cutoff and we cannot

exhibit a counter-example.
Let's see the exploration algorithm graph with the new cutoff definition.



Here, when the algorithm reaches event 6 it does not find a cut-off because neither condition (1) nor condition (2) is satisfied. ($\#_F([e_6]) = |\{q_0\}|+|\{q_0\}| = 2\rangle \#_F([e_5]) = |\{q_0\}|$) Thus the exploration continues and will find event 3 which is declared as cutoff with event 5 as corresponding event. Here, we apply our first solution for cut-off in the same branch and we obtain an illegal infinite-trace $\langle s_0, q_0\rangle \xrightarrow{b} \langle s_1, q_0\rangle (\xrightarrow{d} \langle s_1, q_0\rangle \xrightarrow{c} \langle s1, q_0\rangle)^\omega \in (P \otimes A_{\neg\phi})$. Thus $P \not\models \phi$.

4 A new Unfolding based partial order reduction approach to LTL Model Checking

In section 3 , we bring solutions to the issues caused by cutoff in UDPOR. They are mainly based on a refined notion of cutoff. However, we still need to fix one thing : the detection of livelocks.
To detect livelock we will use an algorithmic solution in our update version of the exploration algorthim of [9]. The idea is as follows: at each step of the exploration we maintain a counter hv that represents the last visible discovered event's height.

Hence, when we will detect a cutoff event (in the same branch) $e = (a_p, H, (q_1, x_1, q_1'))$ and his correspondent $e' = (a_p', H', (q_2, x_2, q_2'))$ we will check if $x_1 = \epsilon$ that's mean the program action is done on invisible actions and if $hv\langle H'$ which mean that the correspondent is an invisible event too i.e a livelock .

Now we have all elements to change the exploration algorithm of [9] and verify LTL formulae with UDPOR.

Algorithm 1 LTL Model Checking with UDPOR

procedure EXPLORE(C,D,A,hv):
Extend(C,hv)
if $en(C) = \emptyset$ **then**
  return
end if
if $A = \emptyset$ **then**
  Choose $e = \langle a, H, q\rangle \in en(C)$
else
  Choose $e = \langle a, H, q\rangle \in en(C) \cap A$
end if
if $label(a) \neq \epsilon$ **then**
  $hv = hv + 1$
end if
Explore(C,D,A,hv)
if $\exists J \in Alt(C, D \cup \{e\}$ **then**
  Explore(C,$D \cup \{e\}$,$J \setminus C$)
end if
Remove(e,C,D)

Algorithm 2 Extend(C,hv)

for $e = \langle e, H, q\rangle \in ex(C)$ **do**
  $(e' = \langle a', H', q'\rangle, found, type) = cutoff(e)$
  if $found$ **then**
    if $label(a) = \epsilon$ & $hv < |H'|$ & $type = 1$ **then**
      livelock!
    end if
  else
    $U = U \cup \{e\}$
  end if
end for=0

Algorithm 3 cutoff($e = \langle a, H, q \rangle$)

for $e' = \langle e', H', q' \rangle \in U$ **do**
  if $e' \in H$ & $state(e') = state(e)$ & $|H'| < |H|$ **then**
    if $\exists TerminalState \in H \setminus H'$ **then**
      raise exception: illegal infinite trace!
    end if
    return $(e', found = true, type = 1)$
  end if
  if $\neg(e' \in H)$ & $state(e) = state(e')$ & $\#_F(H') \geq \#_F(H)$ **then**
    return $(e', found = true, type = 2)$
  end if
end for
return $(null, found = false)$
=0

What has been mainly changed in our version of the exploration algorithm are the Cutoff() and Extend() functions. For the Cutoff() function we modify the condition presented in section 3 into the new one. The function Extend() adds every new event enabled if neither a cutoff is detected nor a livelock is raised. To detect these livelocks we add the counter hv presented at the beginning of this section which incremented every time a visible event is picked in the main procedure. Then a cutoff between two invisible events (in the same branch) is detected and hv is smaller than the history of the representative, that means we have a livelock here.

5 Conclusion and Future work

We have proposed an extended version of the Unfolding-based Dynamic Partial Order Reduction with a refined definition of cutoff inspire from [2] to handle the model checking of LTL. This new definition of cutoff allows us to capture loops that are represented by events in different branches of the tree explored by the exploration algorithm which was not possible with the original one. In the future, we aim at applying this to an action-based Linear Temporal Logic and implementing this UDPOR algorithm to experiment it and compare it with state-of-the-art tools. Other works such as adapting the new algorithm to another programming models like the one in [6] are considered.

6 References

1. Albert, E., Gomez-Zamalloa, M., Isabel, M., Rubio, A.: Constrained Dynamic Partial Order Reduction. In: 30th International Conference on Computer Aided Verification, CAV'18, Oxford, UK. pp. 392–410 (July 2018). https://doi.org/10.1007/9

2. Aronis, S., Jonsson, B., Lang, M., Sagonas, K.: Optimal dynamic partial order reduction with observers. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS'18. pp. 229–248. Springer (2018). https://doi.org/10.1007/978-3-319-89963-314

3. Esparza, J., Heljanko, K.: Unfoldings - A Partial-Order Approach to Model Checking. Monographs in Theoretical Computer Science. An EATCS Series, Springer (2008). https://doi.org/10.1007/978-3-540-77426-6

4. Esparza, J., Heljanko, K.: A new unfolding approach to LTL model checking. In: Montanari, U., Rolim, J.D.P., Welzl, E. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 475–486. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45022-X

5. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'05, Long Beach, California, USA. pp. 110–121 (January 2005). https://doi.org/10.1145/1040305.1040315.

6. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, Lecture Notes in Computer Science, vol. 1032. Springer (1996). https://doi.org/10.1007/3-540-60761-7

7. McMillan, K.L., "Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits," in Proceedings of the 4th Workshop on Computer Aided Verification, Montreal, LNCS, Vol. 663, 1992, pp. 164-174.

8. Pham,T.A., Jéron T., Quinson M. (2019) Unfolding-based dynamic partial order reduction of asynchronous distributed programs. In: Pérez JA, Yoshida N (eds) Formal techniques for distributed objects, components, and systems—39th IFIP WG 6.1 international conference, FORTE 2019, held as part of the 14th international federated conference on distributed computing techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings, Springer, Lecture Notes in Computer Science, vol 11535, pp 224–241. https://doi.org/10.1007/978-3-030-21759-4_13.

9. Rodriguez, C., Sousa, M., Sharma, S., Kroening, D.: Unfolding-based partial order reduction. In: 26th International Conference on Concurrency Theory, CONCUR'15, Madrid, Spain. pp. 456–469 (September 2015). https://doi.org/10.4230/LIPIcs.CONCUR.2015.456.

10. Nguyen, H.T.T., Rodríguez, C., Sousa, M., Coti, C., Petrucci, L.: Quasi-optimal partial order reduction. CoRR, abs/1802.03950 (2018)