Partial Order Reduction for Branching Security Protocols

W. Fokkink [1,2], M. Torabi Dashti [2], and A. Wijs [2]

[1] Vrije Universiteit, Amsterdam
[2] CWI, Amsterdam

Abstract. We extend the partial order reduction algorithm of Clarke et al. [CJM00] to handle branching security protocols, such as optimistic fair exchange protocols. Applications of the proposed algorithm in both model checking and constraint solving approaches are discussed. We also report some experimental results using a μCRL implementation of the algorithm.

1 Introduction

Two main approaches to automatic verification of security protocols are model checking and constraint solving. Both these techniques in principle need to enumerate all possible interleavings of actions performed by protocol participants. Partial order reduction (POR) techniques identify and avoid generating identical interleavings, modulo the properties that are to be verified, to reduce the time and memory used in the verification procedure. Clarke et al. [CJM00] were the first to formally present a POR algorithm for security protocols and determine the class of modal properties that are preserved by it. They observe that the intruder's knowledge in the course of a protocol run is non-decreasing and, intuitively, with more knowledge the intruder can do more (harm). Therefore, when verifying reachability properties, it is safe to prioritise actions which increase the intruder's knowledge over other actions. This is the heart of the POR algorithm of [CJM00]. This algorithm was originally used in BRUTUS, a tailored model checker for verifying security protocols, and, later on, in [MS01] to reduce the number of constraint sets that have to be solved in order to verify a security protocol.

Motivations The POR algorithm of [CJM00] assumes that security protocols are non-branching. Each participant of a non-branching protocol at each state of the protocol has at most one single action to perform, i.e. no alternative options are available to it. These protocols are widely used in the literature as an abstract model for various authentication and key distribution schemes since early years of security protocol analysis (e.g. see "ping-pong protocols" of [DEK82]). However, in practice, participants of, for example, authentication protocols have choice points in the course of a protocol run. They can for instance take alternative steps when a received message does not satisfy its preset conditions or a timeout occurs. Therefore, any faithful model of these protocols needs to allow such choice points in the specification as well. More importantly, some security protocols are inherently branching, i.e. they cannot be modelled properly without allowing branching protocol participants. Prominent examples of branching security protocols are optimistic fair exchange protocols, including non-repudiation, fair payment, certified email and electronic contract signing protocols (e.g. see [Aso98]). Participants of these protocols can choose between continuing the normal flow of the protocol and resorting to trusted third parties, in case of long delays of the opponent or communication failures. As is motivated in this paper, branching behaviours of participants require special considerations when performing POR.

Contributions We extend the POR algorithm of Clarke et al. to handle branching security protocols. To achieve this, we present an enriched model of security protocols that explicitly allows conditionals and choice points in the specification of participants. We first focus on how the extended POR algorithm can be used in explicit state model checking approaches to security protocol verification. Next we discuss the application of the proposed POR to constraint solving for security protocol verification. Some experimental results using a μCRL [BFG$^+$01] implementation of the extended POR algorithm are also presented in the paper.

Related work Algorithms for POR are by now an established branch of model checking and state space generation fields, see, e.g., [PPH96,Pel98,CGP00]. POR techniques for analysing security protocols can perhaps be traced back to [SS98], where some methods to reduce the number of states when verifying security protocols are proposed, but no formalisation of the techniques is provided. Similarly, [Bas99] applies heuristics to prune the search space of security protocols, but only intuitively justifies them. We follow the approach of [CJM00] that formally presents a POR algorithm for security protocols. In [CM05], a POR algorithm for security protocol has been proposed that preserves a limited class of properties compared to [CJM00], but gains more reductions. We did not consider extending the algorithm of [CM05], because, first, the logic that is preserved by this algorithm is rather restricted. Second, it heavily relies on the depth-first traversal strategy and cannot easily be embedded in a breadth-first exploration algorithm. It would thus defy an efficient distributed implementation, which is part of our future plans. Both [CJM00] and [CM05] have implemented their POR algorithms in special-purpose tools for model checking security protocols. In contrast, we have incorporated our POR algorithm in a general purpose flexible state space generation and reduction framework based on the process algebra μCRL [GP95].

Structure of the paper We start in section 2 with introducing some preliminary concepts. In section 3.1, we shortly recall the algorithm of [CJM00], and then we introduce our POR algorithm for branching security protocols in section 3.2. In appendix A we prove that the proposed POR algorithm preserves LTL_{-X} properties, by showing that the reduced state space is stuttering equivalent to the full state space. We mainly focus on how the proposed POR algorithm can be used in explicit state model checking. The case of constraint solving is discussed in section 4. In section 5 we report some experimental results using our POR algorithm. Finally we conclude the paper in section 6.

2 Preliminaries

We consider a model of processes with asynchronous message passing via a network which is controlled by the attacker. First we set up the machinery required to formalise this model.

2.1 Messages

We model security protocols as a collection of *message* passing processes. Below we define the set of messages.

Definition 1. *Let MF be a set of function symbols and MV be a set of message variables. The set of messages Msg is defined as:*

- $m \in MV \implies m \in Msg$
- $m_1, \cdots, m_k \in Msg \implies f(m_1, \cdots, m_k) \in Msg$*, if f is a k-ary function symbol in MF. For function symbols of arity* 0 *(serving as constants) instead of $f()$ we write f.*

Let the function $var : Msg \rightarrow 2^{MV}$ return all message variables occurring in a message. We define $var(\{m\} \cup M) = var(m) \cup var(M)$ and $var(\emptyset) = \emptyset$. A messages m is called *closed* iff $var(m) = \emptyset$. The set of all closed messages is $Msg^c = \{m \in Msg \mid var(m) = \emptyset\}$. Let σ be a partial function $\sigma : MV \rightarrow Msg^c$. For a message $m \in Msg$, $[m]_\sigma$ is the message that is obtained by simultaneously substituting all message variables $v \in var(m)$ in m with their corresponding $\sigma(v)$. The domain of partial function σ is denoted by $d(\sigma)$.

2.2 Labelled transition systems

A Labelled Transition System (LTS) is a tuple (Σ, s_0, Act, Tr), where Σ is a set of states, $s_0 \in \Sigma$ is the initial state, Act is a set of actions and $Tr \subseteq \Sigma \times Act \times \Sigma$ is the transition relation. A transition $(s, a, s') \in Tr$, denoted $s \xrightarrow{a} s'$, indicates that the system can move from state s to s' by performing action a.

Each action $a \in Act$ is a pair $a = (l, m)$, also denoted $l(m)$, where l is the action's *label* and $m \in Msg$ is the action's *parameter*. The set of all action labels of Act is denoted $\Lambda_{Act} = \{l \mid \exists a \in Act, m \in Msg.\ a = (l, m)\}$. The action set Act is called *closed* iff $var(\Omega_{Act}) = \emptyset$, where $\Omega_{Act} = \{m \in Msg \mid \exists a \in Act, l \in \Lambda_{Act}.\ a = (l, m)\}$.

The set of enabled actions in state s is $en(s) = \{a \in Act \mid \exists s' \in \Sigma.\ s \xrightarrow{a} s'\}$. For $A \subseteq Act$, we let $nxt(s, A) = \{s' \in \Sigma \mid \exists a \in A.\ s \xrightarrow{a} s'\}$. We write $\rightarrow^*$ for the reflexive transitive closure of $\xrightarrow{a}$ for any $a \in Act$.

Now we are ready to describe different properties of LTSs. An LTS is called *finite* if its set of states Σ and its set of actions Act are finite. An *acyclic* LTS is such that $\forall s, s' \in \Sigma, a \in Act.\ s \xrightarrow{a} s' \implies \neg\ (s' \rightarrow^* s)$. A *branching* (or *deterministic*) LTS is such that $\forall s \in \Sigma, a \in Act.\ (s \xrightarrow{a} s' \ \wedge\ s \xrightarrow{a} s'') \implies s' = s''$. A *non-branching* LTS is such that $\forall s \in \Sigma, a, a' \in Act.\ (s \xrightarrow{a} s' \wedge s \xrightarrow{a'} s'') \implies (s' = s'' \wedge a = a')$.

A *property* ϕ is a function $\phi : \text{Ł} \rightarrow \{\mathsf{T}, \mathsf{F}\}$, where Ł is the set of all LTSs. Given a set of properties Φ and two LTSs L and L', we write $L \simeq_\Phi L'$ iff $\forall \phi \in \Phi.\ \phi(L) = \phi(L')$.

2.3 Modelling security protocols

Our model of security protocols is related to the strand space formalism of [JHG99]. We model a security protocol as a finite number of processes and an asynchronous communication subsystem. The set of protocol participants is divided into a set of *honest* processes, denoted by P, and one single intruder process which models all malicious participants and, moreover, controls the communication subsystem. We consider the Dolev-Yao (DY) model [DY83] as the intruder. Each process $p \in P$ is an LTS in itself, with the annotation that it can interact with the communication subsystem. From this point on, the two terms process and LTS are used interchangeably. We require that the processes that model roles of honest participants of the protocol are finite, acyclic, deterministic and uniquely named. We first describe the behaviour of honest processes.

To interact with the communication subsystem, a process $p \in P$ has two designated actions $send_p(m)$ and $recv_p(m)$, in which message m is produced and consumed, respectively. Apart from *send* and *recv*, [1] all other actions of honest processes are assumed internal, i.e. not interacting with the communication subsystem. These are symbolic actions which typically denote security claims of protocol participants or their internal decisions. An internal action is called *invisible* if it does not appear in the properties being verified. Else, it is called *visible*. We assume that all internal actions of process p contain p as a subscript, e.g. $secret_p(m)$ can be an internal action performed by p when concluding that m is a secret. Note that internal actions can also have messages as parameters, as is shown in the aforementioned example. The set of action labels of a process $p = (\Sigma_p, s_{0_p}, Act_p, Tr_p)$ can thus be partitioned into four groups: $\Lambda_{Act_p} = V_p \cup I_p \cup S_p \cup R_p$, with V_p and I_p denoting the set of visible internal actions and the set of invisible internal actions of p, respectively, $S_p = \{send_p\}$ and $R_p = \{recv_p\}$. Since all these actions are subscripted with their corresponding process names, for any two different honest processes p and q, we have $Act_p \cap Act_q = \emptyset$. To avoid name clashes, we assume that $\forall p,q \in P.\ p \neq q \implies var(\Omega_{Act_p}) \cap var(\Omega_{Act_q}) = \emptyset$.

Below we define *well-formed* processes. Intuitively, a well-formed process can only send (or decide based on) closed messages in the course of any protocol execution. But before that, we need to introduce a notion of *path*.

Given a process $p = (\Sigma_p, s_{0_p}, Act_p, Tr_p)$, a path (or execution) in p is a sequence $\xi = s_0, \alpha_0, s_1, \alpha_1, \ldots$, where $s_i \in \Sigma_p$ and $\alpha_i \in Act_p$, such that $(s_i, \alpha_i, s_{i+1}) \in Tr_p$, for $i \geq 0$. When $s_0 = s_{0_p}$, ξ is called rooted.

Definition 2. *A process p is called well-formed iff the following property holds in any of its rooted paths $s_0, \alpha_0, s_1, \alpha_1, \cdots$: For any $\alpha_i = l_p(m)$, if $l_p \in V_p \cup I_p \cup S_p$, then $var(m) \subseteq \cup_{\{m' \mid \exists j<i.\ \alpha_j = l'_p(m')\}} var(m')$.*

We require all members of P (i.e. honest participants) to be well-formed processes. However, no such condition is put on the intruder process.

Each process $p \in P$ has a special set of action labels $B_p \subseteq I_p$ that model its internal choices. These action labels behave as Boolean functions and can affect the execution flow of the process, i.e. p may use the results on these functions to decide which branch to follow. Therefore, for each action label $l_p \in B_p$, we have $l_p : Msg^c \to \{\mathsf{T}, \mathsf{F}\}$.

Intruder model To model the DY intruder, which has complete control over communication channels, we assume it plays the role of the communication media. All messages are thus channelled through the intruder: Even though process p sends the message m

[1] We conventionally omit the subscript p from $send_p$ and $recv_p$ when this is clear from the context or the discussion does not depend on any particular process performing these actions.

with the intention that it should be received by process q, it is in fact the intruder that receives the message from p, and it is from the intruder that q can receive m. The DY intruder process is denoted by $DY = (\Sigma_{DY}, s_{0_{DY}}, Act_{DY}, Tr_{DY})$.

The process DY is always ready to receive messages from other processes, as it plays the role of the communication subsystem. It subsequently adds the received messages to its knowledge, that is basically a set of messages. We define the function $K : \Sigma_{DY} \to 2^{Msg^c}$ to return the set of intruder knowledge at a given intruder state. We assume that the state of the intruder is uniquely described with its knowledge set, i.e. K is injective.

The DY intruder can also send a message if it can *synth*esize it from its knowledge. We do not explicitly define the function $synth : Msg^c \times 2^{Msg^c} \to \{\mathsf{T}, \mathsf{F}\}$, as our results do not depend on the rules underlying *synth*, except for its monotonicity: $\forall m \in Msg^c, X, Y \in 2^{Msg^c}.\ X \subseteq Y \implies (synth(m, X) \implies synth(m, Y))$.

System properties We consider *synchronous* communication between honest participants and the intruder model, as is defined below. However, the communication between participants of a protocol, via the DY intruder, is asynchronous (the intruder can be seen as an unbounded message buffer) and a participant has no guarantee about the origin of the messages it receives.

Let $p_1 = (\Sigma_1, s_{0_1}, Act_1, Tr_1), \cdots, p_n = (\Sigma_n, s_{0_n}, Act_n, Tr_n)$ be n honest participants of the protocol, i.e. $P = \{p_1, \cdots, p_n\}$.

Definition 3. *The synchronous product of processes $p_1, \cdots, p_n$ and DY, denoted by $p_1 \| \cdots \| p_n \| DY$, is an LTS $L = (\Sigma, s_0, Act, Tr)$, in which each state $s = \langle x_0, \cdots, x_n, k_{DY}, \sigma \rangle$ contains the state of each process of P, the state of the intruder knowledge and also a partial function $\sigma : MV \to Msg^c$. Below, we describe how L is constructed.*

- *Initial state: $s_0 = \langle s_{0_1}, \cdots, s_{0_n}, K(s_{0_{DY}}), \emptyset \rangle$.*
- *Non-Boolean internal actions:*
  The transition $\langle x_0, \cdots, x_i, \cdots, x_n, k, \sigma \rangle \xrightarrow{l_{p_i}(m)} \langle x_0, \cdots, x_i', \cdots, x_n, k, \sigma \rangle$ can happen iff $\exists m' \in Msg,\ l_{p_i} \in V_{p_i} \cup I_{p_i} \setminus B_{p_i}.\ (x_i, l_{p_i}(m'), x_i') \in Tr_i \ \wedge\ m = [m']_\sigma$.
- *Boolean internal actions:*
  The transition $\langle x_0, \cdots, x_i, \cdots, x_n, k, \sigma \rangle \xrightarrow{\tau_{p_i}} \langle x_0, \cdots, x_i', \cdots, x_n, k, \sigma \rangle$ can happen iff $\exists m' \in Msg,\ l_{p_i} \in B_{p_i}.\ (x_i, l_{p_i}(m'), x_i') \in Tr_i \ \wedge\ l_{p_i}([m']_\sigma) = \mathsf{T}$.
- **send** *actions:*
  The transition $\langle x_0, \cdots, x_i, \cdots, x_n, k, \sigma \rangle \xrightarrow{\mathbf{send}_{p_i}(m)} \langle x_0, \cdots, x_i', \cdots, x_n, k \cup \{m\}$ [2]$, \sigma \rangle$ can happen iff $\exists m' \in Msg.\ (x_i, send_{p_i}(m'), x_i') \in Tr_i \ \wedge\ m = [m']_\sigma$.
- **recv** *actions:*
  The transition $\langle x_0, \cdots, x_i, \cdots, x_n, k, \sigma \rangle \xrightarrow{\mathbf{recv}_{p_i}(m)} \langle x_0, \cdots, x_i', \cdots, x_n, k, \sigma' \rangle$ can happen iff

$$\begin{aligned}&\exists m' \in Msg.\ (x_i, recv_{p_i}(m'), x_i') \in Tr_i \ \wedge \\ &\qquad \exists \sigma'.\ \sigma \subseteq \sigma' \ \wedge\ d(\sigma') \subseteq d(\sigma) \cup var(m') \ \wedge \\ &\qquad\qquad m = [m']_{\sigma'} \ \wedge\ synth(m, k).\end{aligned}$$

[2] To keep the knowledge set K uniform, one can decompose message m into its parts before adding it to K, c.f. [Pau98].

The set of action labels of Act can be partitioned into four disjoint sets as $\Lambda_{Act} = V \cup I \cup S \cup R$, where $V = \cup_{p \in P} V_p$, $I = \cup_{p \in P}(I_p \setminus B_p) \cup \{\tau_p\}$, $S = \cup_{p \in P}\{\mathbf{send}_p\}$ and $R = \cup_{p \in P}\{\mathbf{recv}_p\}$. Note that, for convenience, we choose to write the elements of S and R in bold face (**send** and **recv**, in contrast to *send* and *recv*) to ease referring to the LTS in which context they happen. The action $\mathbf{recv}_p$ is the result of the communication between the intruder process (which we do not explicitly specify) and p performing a $recv_p$ action (and similarly for $\mathbf{send}_p$ and $send_p$).

We remark that performing a $\mathbf{recv}_p(m)$ action depends not only on the state of p, but also on the intruder's ability to construct the message m. Whereas for a **send** to happen, no condition is put on the intruder's state. Similarly, internal actions of a process can happen with no conditions on other processes' or the intruder's states.

It is worth mentioning that since all processes p_i are deterministic and $Act_{p_i} \cap Act_{p_j} = \emptyset$ for $i \neq j$, L is deterministic as well. Besides, as we only consider well-formed processes, Act is closed, that is all members of Act have variable-free messages as parameters. However, we cannot claim that L is finite based on the finiteness of p_i, because the DY process is not necessarily finite. In fact, it can compose an infinite number of messages with consecutively pairing a single constant value (e.g. see [DY83]), hence being infinitely branching. However, usually model checking algorithms hinge on the finiteness of the model, and so does our proposed POR algorithm. Therefore, in the following discussions we assume that L is finite (for a complementary discussion see section 4 on POR for constraint solving approaches). In current practice of model checking security protocols, finiteness of the model can be achieved by, for instance, hard-coding the intruder's message templates in the model [KR03] or assuming typed messages [CCT05]. Moreover, L is acyclic since all p_i are (by definition) acyclic.

2.4 State space generation

A state space generation algorithm is normally provided with the LTSs $p_1, \cdots, p_n$ and DY as input specification, also referred to as spec, and produces $L = p_1 \| \cdots \| p_n \| DY$ (see definition 3), the LTS that is described by putting them in parallel, as output. We denote this procedure by spec $\overset{std}{\Rightarrow} L$.

As an example, algorithm 1 shows a breadth-first state space generation algorithm. This algorithm is guaranteed to terminate when generating acyclic finite LTSs (in our model, security protocols are assumed to result in acyclic finite LTSs, see section 2.3). Here, we confine to the traversal strategy and abstract away generating the output (file). In algorithm 1, sets *Current* and *Next* denote the set of states at the current and the next level, respectively. For the definitions of *en* and *nxt* see section 2.2.

2.5 Partial order reduction

The main principle of POR is to exploit the commutativity of concurrently executed actions in order to generate only a sufficient fraction of the state space. A POR algorithm explores a subset of enabled actions $ample(s) \subseteq en(s)$ at each state s, such that a certain class of desired properties is preserved. For a given specification spec, let L and L' be the results of a standard state space generation *std* and of a POR algorithm *por* on spec

Algorithm 1 Breadth-first state space generation

```
REQUIRES: $p_1, \cdots, p_n, DY$
RETURNS: $p_1 \| \cdots \| p_n \| DY$
  $Current := \{s_0\}$
  $Next := \emptyset$
  while $Current \neq \emptyset$ do
    for all $s \in Current$ do
      $Next := Next \cup\ nxt(s, en(s))$
    end for
    $Current := Next$
    $Next := \emptyset$
  end while
```

respectively: $\mathsf{spec} \overset{std}{\Rightarrow} L$ and $\mathsf{spec} \overset{por}{\Rightarrow} L'$. For the set of properties Φ that is to be preserved by *por*, we require $L \simeq_{\Phi} L'$. For a general introduction to POR see [CGP00].

3 Partial order reduction for security protocols

In this section, first, we briefly describe the POR algorithm of [CJM00]. This is a POR algorithm tailored for verifying reachability properties of non-branching security protocols, such as authentication and key distribution protocols. Next, we extend this algorithm to cover branching security protocols.

3.1 POR for non-branching security protocols

In [CJM00] it is observed that the knowledge of the DY intruder is non-decreasing, and with more knowledge more states are reachable by the intruder. It means that when verifying reachability properties, intuitively **send** actions, which typically increase the intruder's knowledge, can be prioritised over other actions in the state space describing the security protocol. This is the heart of the POR algorithm for non-branching security protocols that is proposed in [CJM00]. The set of actions to be explored at each state s, namely $ample(s)$, is chosen as follows in [CJM00]:

- If $en(s)$ contains an invisible internal action, then $ample(s)$ is a singleton containing an arbitrary invisible action picked from $en(s)$.
- Suppose $en(s)$ does not contain an invisible internal action, but does contain a **send** action. In this case $ample(s)$ is a singleton containing an arbitrary **send** action picked from $en(s)$.
- If $en(s)$ does not contain an invisible action or a **send** action, $ample(s) = en(s)$.

We briefly give an informal reason why **recv** actions cannot be prioritised over other actions. Let $s \xrightarrow{\mathbf{recv}(m_1)} s_1$ and $s \xrightarrow{\mathbf{send}(m_2)} s_2$. Since $K(s) \subseteq K(s_2)$ (see definition 3) and *synth* is monotonic, generally more **recv** actions can be instantiated at s_2. That is why **recv** actions may not be prioritised over **send** actions. A **recv** action cannot be prioritised over other **recv** or internal actions because they might subsequently enable other **send** actions. A formal treatment of this algorithm can be found in [CJM00].

To specify the requirements of security protocols, in [CJM00], a first order logic where quantifiers range over protocol participants, augmented with a past time modal operator is considered. Their POR algorithm is shown to preserve formulae of this logic.

3.2 POR for branching security protocols

Motivations The POR algorithm of [CJM00] is not suitable for branching security protocols. Examples of branching security protocols are optimistic fair exchange, digital contract signing, certified e-mail, and non-repudiation protocols, see e.g. [Aso98,KMZ02]. Participants of these protocols can typically choose between multiple *send*, *recv* and internal actions. These are therefore modelled as finite, acyclic, branching named processes. Taking only one **send** from a process into the *ample* set is not safe in these cases, because it can in principle disable other actions of that process. This motivates extending the POR algorithm of [CJM00] to branching security protocols. Below, we introduce our POR algorithm for branching security protocols. Next, we describe the class of properties that are preserved by the reduction algorithm. We consider an action-based version of LTL_{-X} (see, e.g., [CGP00] for a formal definition of LTL_{-X}). Appendix A shows that a reduced state space using the proposed POR algorithm is stuttering equivalent to the original state space (see, e.g., [CGP00] for a formal definition of stuttering equivalence). This serves as a proof for preserving LTL_{-X} properties. We could in principle prove this result for the logic fragment that is used in [CJM00] as well. However, we chose a more standard logic.

POR algorithm We mainly follow the idea of [CJM00]. Let $L = p_1 \| \cdots \| p_n \| DY$ be described as $L = (\Sigma, s_0, Act, Tr)$, such that $\Lambda_{Act} = I \cup V \cup S \cup R$. We prioritise transitions with action labels of the set $I \cup S$ over those of $R \cup V$. However, since processes can branch in our model, if one action of process p is taken into the *ample* set, all other actions of p enabled at that state should be taken as well. Hence, actions of $I \cup S$ can be prioritised over others only if their corresponding process does not perform any action from $V \cup R$ at that state. This raises a new problem, as is discussed below.

Let p be a process. Consider the following two scenarios: First, the case where p can only perform one $send_p$ action at state s. Second the case where p can perform $send_p$ and $recv_p$ at state s, but the $recv_p$ action of p is not present in the resulting LTS as $\mathbf{recv}_p$, because the intruder does not have enough knowledge to compose that message (see the definition of synchronous product in section 2.3). These two scenarios are represented in exactly the same way in the resulting LTS L, [3] while the POR algorithm needs to differentiate between them. This is because in the first scenario it is safe to prioritise p's action over other actions in s, while in the second scenario it is not. A solution to this problem is to statically pre-process the specification of the protocol by adding a dummy κ action as an always available choice to all *recv* actions in every process specification. [4]

[3] The problem that is discussed here is rather specific to the state space generation setting and is in principle not relevant to constraint solving approaches.

[4] We assume that in the specification, κ is solely used for this purpose.

Definition 4. *Given a process $p = (S_p, s_{0_p}, Act_p, Tr_p)$, a κ-translation of p is an LTS $p^\kappa = (S_p, s_{0_p}, Act_p \cup \{\kappa_p\}, \mathsf{add}_\kappa(Tr_p))$, where add_κ is defined as $\mathsf{add}_\kappa(\emptyset) = \emptyset$ and*

$$\mathsf{add}_\kappa(\{(s,a,s')\} \cup T) = \begin{cases} \{(s,\kappa_p,s'\} \cup \{(s,a,s')\} \cup \mathsf{add}_\kappa(T) & \text{if } a = recv_p(m) \\ \{(s,a,s')\} \cup \mathsf{add}_\kappa(T) & \text{else} \end{cases}$$

The internal actions κ can subsequently be used in the generation phase to detect the existence of non-enabled *recv* actions at each state (i.e. if the corresponding **recv** action is not present in the LTS).

Our POR algorithm can be described as follows. Assume a specification spec comprising a set of processes $p_1, \cdots, p_n, DY$. The *full* state space is defined as spec $\overset{std}{\Rightarrow} L$, that is $L = p_1 \| \cdots \| p_n \| DY$. We define $\mathsf{spec}^\kappa = p_1^\kappa, \cdots, p_n^\kappa, DY$. We let $\mathsf{spec}^\kappa \overset{std}{\Rightarrow} L^\kappa$, that is $L^\kappa = p_1^\kappa \| \cdots \| p_n^\kappa \| DY$. Our POR algorithm is applied on spec^κ and results in L_{por}: $\mathsf{spec}^\kappa \overset{por}{\Rightarrow} L_{por}$.

In the following discussions, we write $L = (\Sigma, s_0, Act, Tr)$ and $L^\kappa = (\Sigma^\kappa, s_0^\kappa, Act^\kappa, Tr^\kappa)$. Observe that $\Sigma^\kappa = \Sigma$, $s_0 = s_0^\kappa$, $Act^\kappa = Act \cup \{\kappa_p \mid p \in P\}$. We partition Λ_{Act^κ} into four action groups: $\Lambda_{Act^\kappa} = V^\kappa \cup I^\kappa \cup S^\kappa \cup R^\kappa$, where $V^\kappa = V$, $I^\kappa = I$, $S^\kappa = S$ and $R^\kappa = R \cup \{\kappa_p \mid p \in P\}$ (the definitions of V, I, S and R are given in section 2.3, the system properties subsection). By adding κ actions to R^κ, we ensure that these actions are treated as (artificial) **recv** actions.

Below, we continue with introducing some notions used in our POR algorithm.

Definition 5. *We define the projection function $\pi : Act^\kappa \to P$ as:*

$$\begin{array}{l} \pi(a) = p, \ \textit{if } a = a_p(m),\ a_p \in V^\kappa \cup I^\kappa \cup S^\kappa \cup R^\kappa \ \textit{ for some } m \in \Omega_{Act^\kappa} \\ \pi(a) = p, \ \textit{if } a = \kappa_p \end{array}$$

Definition 6. *We define the relation $\sim \subseteq Act^\kappa \times Act^\kappa$ as $\forall a, a' \in Act^\kappa.\ a \sim a' \iff \pi(a) = \pi(a')$.*

Clearly $\sim$ is an equivalence relation, and thus partitions Act^κ into equivalence classes such that each class contains only actions performed by one particular process. The set of all equivalence classes in $A \subseteq Act^\kappa$ given the equivalence relation $\sim$ is denoted $A/\sim$. For an action a and a state s, we write $[a]_s = [a] \cap en(s)$, where the equivalence class $[a]$ is defined as $[a] = \{a' \in Act^\kappa \mid a \sim a'\}$.

We define the function $\mathbb{V} : 2^{Act^\kappa} \to 2^{Act^\kappa}$ as $\mathbb{V}(A) = \{a \in A \mid \Lambda_{\{a\}} \subseteq V^\kappa\}$. Intuitively, $\mathbb{V}(A)$ is the largest subset of A such that all its action labels belong to V^κ. The functions $\mathbb{I}$, $\mathbb{S}$ and $\mathbb{R}$ are defined similarly. We let $\partial_\kappa : 2^{Act^\kappa} \to 2^{Act^\kappa}$ be $\partial_\kappa(A) = A \setminus \{\kappa_p \mid p \in P\}$.

Before defining which requirements the *ample* set in POR for branching security protocols has to satisfy, we note that definitions 5 and 6 can trivially be extended to Act (since $Act \subseteq Act^\kappa$), thus being legitimate to be used in L. As mentioned earlier, our POR algorithm receives spec^κ as input, so the conditions which are checked in the algorithm refer to this setting (having L^κ in mind), but the final stuttering equivalence is proved with regard to L as the full state space.

Definition 7. *At each state s, the set of actions to explore, i.e. ample(s), is constructed in two phases. First, we construct an ample$^0(s)$ set that satisfies the following requirements:*

r0. $ample^0(s) \subseteq en(s)$ *and* $ample^0(s) = \emptyset \implies en(s) = \emptyset$.
r1. *For all* $a \in en(s)$, *if* $a \in ample^0(s)$, *then* $[a]_s \subseteq ample^0(s)$.
r2. $\mathbb{V}(ample^0(s)) \neq \emptyset \implies en(s) \subseteq ample^0(s)$.
r3. $\mathbb{R}(ample^0(s)) \neq \emptyset \implies en(s) \subseteq ample^0(s)$.

And in the second phase, the final $ample(s)$ *is defined as* $ample(s) = \partial_\kappa(ample^0(s))$.

Requirement *r0* is a sanity check, c.f. condition C0 in [CGP00]. Requirement *r1* states that if one action of process p is explored, all other enabled actions of p have to be explored as well, since they may disable each other. This requirement was not included in [CJM00] and [CM05] that only deal with non-branching protocols. Requirements *r2* and *r3*, similar to [CJM00], prevent prioritising visible internal actions and **recv** actions over any other action, respectively. As κ actions are merely an artificial apparatus to detect the existence of *recv* actions in case the corresponding **recv** actions are not present in the state space, there is no reason to explore κ actions. Therefore, after constructing $ample^0$, all κ actions are removed from further explorations.

Algorithm 2 shows the construction of an ample set that meets the requirements of definition 7. We emphasise that this algorithm receives spec^κ as its input specification.

Algorithm 2 POR for branching security protocols

```
REQUIRES: $p_1^\kappa, \cdots p_n^\kappa, DY$
RETURNS: $L_{por}$ = a sufficient fragment of $p_1 \| \cdots p_n \| DY$
  $Current := \{s_0\}$
  $Next := \emptyset$
  while $Current \neq \emptyset$ do
    for all $s \in Current$ do
      Construct $en(s)/\sim$ (see definition 6) and name its elements as $c_1, \ldots, c_l$.
      $T := \{c_i, i \in \{1, \ldots, l\} \mid \mathbb{V}(c_i) \cup \mathbb{R}(c_i) = \emptyset\}$
      if $T \neq \emptyset$ then
        Pick a $c \in T$
        $Next := Next \cup nxt(s, c)$
      else if $T = \emptyset$ then
        $Next := Next \cup nxt(s, \partial_\kappa(en(s)))$
      end if
    end for
    $Current := Next$
    $Next := \emptyset$
  end while
```

Logic Below we describe the class of properties that our proposed POR algorithm preserves. We consider an action based version of LTL_{-X}. Given a non-empty set *AP* of atomic propositions on states, the set of formulae of LTL_{-X} is defined inductively as:

– Every member of *AP* belongs to LTL_{-X}.
– If ϕ_1 and ϕ_2 belong to LTL_{-X}, then so do $\neg\phi_1$, $\phi_1 \vee \phi_2$ and $\phi_1 \bigcup \phi_2$.

The usual logical connective are interpreted as usual. Intuitively, the operator $\bigcup$ states that ϕ_2 holds at the current or a future position, and ϕ_1 holds until that position. For a formal definition of the syntax and semantics of LTL_{-X} see, e.g., [CGP00].

To interpret *LTL* formulae, which are originally defined for Kripke structures, on LTSs each state is assigned with the labels of the transitions that sprout out of the state, i.e. each transition label represents an atomic proposition. For a formal treatment of such cross-interpretations see [NV95]. We assume that only actions from $V^{\kappa} \cup \partial_{\kappa}(R^{\kappa})$ may appear as atomic propositions. As we feed the POR algorithm with spec^{κ}, but the full state space is defined with regard to spec, it is worth noting that $V^{\kappa} \cup \partial_{\kappa}(R^{\kappa}) = V \cup R$.

Theorem 1 $L \simeq_{LTL_{-x}} L_{por}$.

Proof. See appendix A.

4 POR in constraint solving

Constraint solving for analysing security protocols has mostly been used in verifying non-branching protocols [MS01]. However, recently the constraint solving approach of [MS01] has been extended to a large class of branching security protocols, namely contract signing protocols [KK05]. In the previous sections we focused on POR algorithms for explicit state model checking settings. Below we sketch how our POR algorithm can be used in the constraint solving setting.

We first cast the constraint solving approaches of [MS01] and [KK05] to our formalism. Given a specification $p_1, \cdots, p_n$, we construct the product of the LTSs $p_1 \times \cdots \times p_n$, where $p \times q$ is defined as the following.

Definition 8. *Given $p = (\Sigma_p, s0_p, Act_p, Tr_p)$ and $q = (\Sigma_q, s0_q, Act_q, Tr_q)$, $p \times q$ is the LTS $(\Sigma_p \times \Sigma_q, (s0_p, s0_q), Act_p \cup Act_q, Tr_{p\times q})$, where $Tr_{p\times q}$ is*

$$\begin{aligned} Tr_{p\times q} = \{&(s_p, s_q) \xrightarrow{a} (s'_p, s_q) \mid (s_p, a, s'_p) \in Tr_p\} \cup \\ \{&(s_p, s_q) \xrightarrow{a} (s_p, s'_q) \mid (s_q, a, s'_q) \in Tr_q\} \end{aligned}$$

Note that p and q do not communicate directly (only indirectly via the intruder). The resulting $p \times q$ can be seen as an uninstantiated state space (i.e. the actions of $Act_{p\times q}$ contain messages with free variables). Moreover, $(p \times q) \times q' = p \times (q \times q')$.

In constraint solving approaches, any rooted path of $p_1 \times \cdots \times p_n$ is examined by a constraint solving algorithm *CS* to decide whether the process *DY* can trick the processes of *P* to execute this path (and, if so, whether it constitutes an attack). We do not specify the *CS* procedure here as our results do not depend on it. We note that our POR algorithm can readily be used in generating a sufficient fragment of the uninstantiated state space $p_1 \times \cdots \times p_n$, see algorithm 3. Constraint solving approaches usually adopt the depth-first traversal strategy, as opposed to the breadth-first strategy of algorithm 3 which conforms to the previous algorithm of this paper. However, the presented algorithm is only a proof of concept and can easily be converted to follow a depth-first traversal strategy.

Algorithm 3 POR for branching security protocols: Constraint solving framework

```
REQUIRES: $p_1, \cdots p_n$
RETURNS: a sufficient fragment of $p_1 \times \cdots \times p_n$
  $Current := \{\langle s_{0_1}, \cdots, s_{0_n} \rangle\}$
  $Next := \emptyset$
  while $Current \neq \emptyset$ do
    for all $s = \langle x_1, \cdots, x_n \rangle \in Current$ do
      Let $c_i = en(x_i)$
      $T := \{c_i, i \in \{1, \ldots, n\} \mid \mathbb{V}(c_i) \cup \mathbb{R}(c_i) = \emptyset\}$
      if $T \neq \emptyset$ then
        Pick a $c_j \in T$
        $Next := Next \cup \{\langle x_1, \cdots, x'_j, \cdots, x_n \rangle \mid x'_j \in nxt(x_j, c_j)\}$
      else if $T = \emptyset$ then
        $Next := Next \cup (\cup_{1 \leq i \leq n} \{\langle x_1, \cdots, x'_i, \cdots, x_n \rangle \mid x'_i \in nxt(x_i, c_i)\})$
      end if
    end for
    $Current := Next$
    $Next := \emptyset$
  end while
```

We remark that in this scenario there is no need to worry about the infinitely branching behaviour of the *DY* process in the generation phase of the uninstantiated state space. Restricting the intruder's behaviour is a task for the *CS* phase in [MS01] and [KK05]. For a similar reason, calculating κ-translations can altogether be omitted from the POR algorithm in this setting. This is because no $recv_p$ action will be hidden as a result of the intruder's lack of knowledge: The intruder's abilities are modelled in the *CS* phase in [MS01] and [KK05].

5 POR experimental results

This section reports some experimental results using a μCRL [BFG$^+$01] implementation of algorithm 2. To demonstrate the effectiveness of the proposed POR algorithm we have modelled a small instantiation of a Digital Rights Management (DRM) protocol, thoroughly described in [JND06]. Below we shortly describe this protocol and our experimental results.

The DRM protocol of [JND06] comprises a finite set of compliant content rendering devices C, a finite set of trusted entities T and an intruder that controls the communication channels. The goal of the protocol is to provide a secure environment for fair exchange of digital items among the members of C. In case of a malicious act (or failure), the suffered party resorts to one of the trusted entities. The set D contains the digital items available in the protocol. Each item is bundled with a right declaring the terms of use of that particular item. The set of rights is denoted R. To keep the state space finite, each $c \in \mathsf{C}$ only has access to a finite set of fresh nonces N_c to start new sessions. Below we consider a small instantiation of this protocol with $|\mathsf{C}| = 2$, $|\mathsf{N}_c| = 1$ for each member of C, and $|\mathsf{D}| = |\mathsf{R}| = 1$. The following table compares the number of states generated and the amount of time consumed by the standard μCRL state space generator and its modification following algorithm 2, for different values of $|\mathsf{T}|$. [5]

Instance	Standard		POR		Reduction
$\lvert\mathsf{T}\rvert$	# States	Time	# States	Time	in the # States
1	4,036	00:06.97	1,253	00:04.16	69.0%
2	92,976	04:02.61	15,124	00:47.15	83.7%
3	2,674,940	15:08:40.86	258,569	15:41.58	90.3%

Although POR loads the generation algorithm with some book-keeping and extra computations, the gained reduction definitely compensates for these costs, as is evident from the time columns. These results unfortunately cannot readily be compared with the existing tools implementing POR for security protocols [CJM00,CM05]. This is because, as mentioned earlier, they do not deal with branching security protocols. Going back to non-branching protocols, our algorithm coincides with the algorithm of [CJM00] (except that as exploration strategy we opted for breadth-first, while [CJM00] chose depth-first). We expect the algorithm of [CM05] to yield more reductions, compared to our algorithm, when analysing non-branching protocols. This is because the algorithm in [CM05] has been optimised for a rather narrow subset of LTL_{-X}, which is the class of properties preserved by our POR algorithm.

6 Conclusions

In this paper we have extended the POR algorithm of [CJM00] to branching security protocols. The proposed algorithm has been implemented in the μCRL general-purpose state space generation toolset, which is based on process algebra.

As future work, we plan investigating two possible extensions of the presented algorithm: First, usually branching security protocols, e.g. fair exchange protocols, have requirements beyond LTL_{-X}. For instance, in [KR03] the requirements of these protocols have been encoded in Alternating-Time Logic [AHK97], and [CD06] shows that some fairness constraints needed when verifying these protocols are not expressible in *LTL*. We thus observe that a POR algorithm that goes beyond *LTL* would be more suitable for these protocols. As the next step, we intend to extend our results to an equivalence relation finer than stuttering equivalence (e.g. failure equivalence [Gla93]).

Second, we plan to implement the proposed POR algorithm in the distributed μCRL toolset [BCL$^+$07]. This seems to be straightforward as the algorithm performs only local tests and no POR-specific inter-workstation communications would be required.

Acknowledgements We are grateful to Jaco van de Pol for his comments on earlier versions of this paper.

[5] These experiments have been performed on a single machine with a 64 bit Athlon 2.2 GHz CPU and 1 GB RAM, running SUSE Linux 9.2.

References

[AHK97] R. Alur, T. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *FOCS*, pages 100–109. IEEE, 1997.

[Aso98] N. Asokan. *Fairness in electronic commerce*. PhD thesis, University of Waterloo, 1998.

[Bas99] D. Basin. Lazy infinite-state analysis of security protocols. In *CQRE*, volume 1740 of *LNCS*, pages 30–42. Springer, 1999.

[BCL$^+$07] S. Blom, J. Calame, B. Lisser, S. Orzan, J. Pang, J. van de Pol, M. Torabi Dashti, and A. Wijs. Distributed analysis with μCRL in practice (tool paper). In *TACAS '07*, 2007. to appear.

[BFG$^+$01] S. Blom, W. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. μCRL: A toolset for analysing algebraic specifications. In *CAV'01*, volume 2102 of *LNCS*, pages 250–254, 2001.

[CCT05] J. Cederquist, R. Corin, and M. Torabi Dashti. On the quest for impartiality: Design and analysis of a fair non-repudiation protocol. In *ICICS '05*, volume 3783 of *LNCS*, pages 27–39. Springer, 2005.

[CD06] J. Cederquist and M. Torabi Dashti. An intruder model for verifying liveness in security protocols. In *FMSE '06*, pages 23–32. ACM Press, 2006.

[CGP00] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[CJM00] E. Clarke, S. Jha, and W. Marrero. Partial order reductions for security protocol verification. In *TACAS'00*, volume 1785 of *LNCS*, pages 503–518, 2000.

[CM05] C. Cremers and S. Mauw. Checking secrecy by means of partial order reduction. In *SAM '04*, volume 3319 of *LNCS*, pages 177–194. Springer, 2005.

[DEK82] D. Dolev, S. Even, and R. Karp. On the security of ping-pong protocols. In *CRYPTO '82*, pages 177–186, 1982.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. on Information Theory*, IT-29(2):198–208, 1983.

[Gla93] R. van Glabbeek. The linear time - branching time spectrum II: The semantics of sequential systems with silent moves. In *CONCUR '93*, volume 715 of *LNCS*, pages 66–81. Springer, 1993.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer, 1995.

[JHG99] F. Javier Thayer Fabrega, J. Herzog, and J. Guttman. Strand spaces: proving security protocols correct. *J. Comput. Secur.*, 7(2-3):191–230, 1999.

[JND06] H. Jonker, S. Krishnan Nair, and M. Torabi Dashti. Nuovo DRM paradiso: formal specification and verification of a DRM protocol. Technical Report SEN-R0602, CWI, Amsterdam, The Netherlands, 2006.

[KK05] D. Kähler and R. Küsters. Constraint solving for contract-signing protocols. In *CONCUR '05*, volume 3653 of *LNCS*, pages 233–247. Springer, 2005.

[KMZ02] S. Kremer, O. Markowitch, and J. Zhou. An intensive survey of non-repudiation protocols. *Computer Communications*, 25(17):1606–1621, 2002.

[KR03] S. Kremer and J.F. Raskin. A game-based verification of non-repudiation and fair exchange protocols. *Journal of Computer Security*, 11(3):399–430, 2003.

[MS01] J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *ACM Conference on Computer and Communications Security*, pages 166–175, 2001.

[NV95] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *J. ACM*, 42(2):458–487, 1995.

[Pau98] L. Paulson. The inductive approach to verifying cryptographic protocols. *J. Computer Security*, 6(1-2):85–128, 1998.
[Pel98] D. Peled. Ten years of partial order reduction. In *CAV '98*, volume 1427 of *LNCS*, pages 17–28. Springer, 1998.
[PPH96] D. Peled, V. Pratt, and G. Holzmann, editors. *Partial order methods in verification*, volume 29 of *DIMACS series in dicrete mathematics and theoretical computer science*. AMS, 1996.
[SS98] V. Shmatikov and U. Stern. Efficient finite-state analysis for large security protocols. In *CSFW '98*, pages 106–115. IEEE Computer Society, 1998.

A Stuttering equivalence

In this section we show that for branching security protocols any POR algorithm that meets the requirements of definition 7 yields a reduced state space that is stuttering equivalent to the full state space. We start with some definitions.

Definition 9. *An* independence *relation $IND \subseteq Act^{\kappa} \times Act^{\kappa}$ is the largest symmetric, anti-reflexive relation, such that for each state s and each $(a,a') \in IND$, if $s \xrightarrow{a} s_1$ and $s \xrightarrow{a'} s'_1$, then $\exists s_2.\, s_1 \xrightarrow{a'} s_2 \wedge s'_1 \xrightarrow{a} s_2$.*

The *dependence* relation *DEP* is defined as $DEP = (Act^{\kappa} \times Act^{\kappa}) \setminus IND$. Two actions a and a' are called *dependent* iff $(a,a') \in DEP$.

In the following, we prove some lemmas about the full state space $L = (\Sigma, s_0, Act, Tr)$, which are used in proving the stuttering-equivalence theorem.

Lemma 1. *For any two actions a and a', we have $[a] \neq [a'] \implies (a,a') \in IND$.*

Proof. Since $\pi(a) \neq \pi(a')$, clearly a and a' do not disable each other. Besides, if $s \xrightarrow{a.a'} s'$ and $s \xrightarrow{a'.a} s''$, then $s' = s''$. This is because performing a has no effect on the state of the process that performs a' and vice versa, as processes do not directly communicate with each other, but only via the intruder (see definition 3). □

Lemma 2. *Assume $\mathbb{R}([a]_s) = \emptyset$ in L^{κ}. Then in L, $s \xrightarrow{a'} s'$ with $[a] \neq [a']$, implies $[a]_s = [a]_{s'}$.*

Proof. Consider L. Since $\pi(a) \neq \pi(a')$, according to lemma 1, $\forall b \in [a].\ (b,a') \in IND$, and consequently, for any $b \in [a]$, $s \xrightarrow{b} s_1$ and $s \xrightarrow{a'} s'$ imply that $\exists s_2.\ s' \xrightarrow{b} s_2$. Therefore $[a]_s \subseteq [a]_{s'}$. Now, we need to show that $[a]_{s'} \subseteq [a]_s$. Note that the enabledness of actions of V, I and S only depend on the state of the process performing them (see definition 3). As the state of $\pi(a)$ is the same in s and s' (because $s \xrightarrow{a'} s'$ and $\pi(a) \neq \pi(a')$), $\mathbb{V}([a]_s) = \mathbb{V}([a]_{s'})$ and similarly for $\mathbb{I}$ and $\mathbb{S}$. However, actions of R also depend on the state of the intruder, i.e. when a process is waiting to receive a message, the intruder's knowledge determines what messages, if any, can be sent to that process. Here, $\mathbb{R}([a]_s) = \emptyset$ in L^{κ} implies that in particular $\kappa_p \notin en(s)$ in L^{κ} (recall that $\forall p \in P.\ \kappa_p \in R^{\kappa}$ in L^{κ}). Therefore, κ_p is not enabled at s' in L^{κ}, as well. As a result, $\mathbb{R}([a]_{s'}) = \emptyset$ in L^{κ}. This implies that also in L, $\mathbb{R}([a]_{s'}) = \emptyset$, completing the proof. □

Proposition 1. *Let $a \in en(s)$ (in L as well as L^κ) and assume that $\mathbb{R}([a]_s) = \emptyset$ in L^κ. For all paths $\xi = \sigma_0, \alpha_0, \sigma_1, \alpha_1, \cdots$ originating from s (i.e. $\sigma_0 = s$) in L, we have:*

$$\forall i.\ (a, \alpha_i) \in DEP \implies \exists j \leq i.\ \alpha_j \in [a]_s.$$

Proof. Let $(\alpha_i, a) \in DEP$, for some i. According to lemma 1, $(a, \alpha_i) \in DEP \implies \alpha_i \in [a]$. Two cases are possible here:

- If $i = 0$: Then obviously letting $j = i$ completes the proof.
- If $i > 0$: Assume $\forall j < i.\ \alpha_j \notin [a]_s$. We prove that then $\alpha_i \in [a]_s$. The assumption in particular implies that $\alpha_0 \notin [a]_s$. Since $\alpha_0 \in en(s)$, we deduce that $\alpha_0 \notin [a]$. Therefore, according to lemma 1, $(a, \alpha_0) \in IND$. As $\mathbb{R}([a]_s) = \emptyset$ in L^κ, lemma 2 implies that $[a]_s = [a]_{\sigma_1}$ in L. Repeating this argument, we can show that $[a]_{\sigma_1} = [a]_{\sigma_2}, \ldots$, and finally $[a]_{\sigma_{i-1}} = [a]_{\sigma_i}$ in L. Hence $[a]_s = [a]_{\sigma_i}$. Since $\alpha_i \in [a]$, and clearly $\alpha_i \in [a]_{\sigma_i}$, it follows that $\alpha_i \in [a]_s$.

□

We recall the following theorem for general LTSs from [CGP00].

Theorem 1. *If the ample set of a POR algorithm satisfies the following properties, then the POR algorithm preserves LTL_{-X}.*

C0. *$ample(s) = \emptyset$ iff $en(s) = \emptyset$.*
C1. *Along every path in the full state space that starts at s, the following condition holds: An action that is dependent on an action in $ample(s)$ cannot be executed without an action in $ample(s)$ occurring first.*
C2. *If $ample(s) \neq en(s)$, then every $a \in ample(s)$ is "invisible", i.e. not appearing in the properties being verified.*
C3. *A cycle is not allowed if it contains a state in which some action a in enabled, but is never included in $ample(s)$ for any state s on the cycle.*

Now we are ready to prove the main theorem about our POR algorithm.

Theorem 2. *L and L_{por} are stuttering equivalent.*

Proof. We show that the conditions C_0, C_1, C_2 and C_3 of theorem 1 hold for our proposed *ample* set:

- Condition C_0 holds because of r_0.
- Condition C_1 holds because of proposition 1.
- Condition C_2 holds because of r_2 and r_3. Recall that we assume that only members of $V^\kappa \cup \partial_\kappa(R^\kappa)$, that is equivalent to $V \cup R$, appear in the properties being verified.
- Condition C_3 holds simply because we consider acyclic security protocols.

□