An Unfolding Algorithm for Synchronous Products of Transition Systems*

Javier Esparza, Stefan Römer

Institut für Informatik, Technische Universität München
e-mail: {esparza,roemer}@in.tum.de

Abstract. The unfolding method, initially introduced for systems modelled by Petri nets, is applied to synchronous products of transition systems, a model introduced by Arnold [2]. An unfolding procedure is provided which exploits the product structure of the model. Its performance is evaluated on a set of benchmarks.

1 Introduction

The unfolding method is a partial order approach to the verification of concurrent systems introduced by McMillan in his Ph. D. Thesis [6]. A finite state system, modelled as a Petri net, is *unfolded* to yield an equivalent acyclic net with a simpler structure. This net is usually infinite, and so in general it cannot be used for automatic verification. However, it is possible to construct a *complete finite prefix* of it containing as much information as the infinite net itself: Loosely speaking, this prefix already contains all the reachable states of the system. The prefix is usually far smaller than the state space, and often smaller than a BDD representation of it, and it can be used as input for efficient verification algorithms. A rather complete bibliography on the unfolding method, containing over 60 papers on semantics, algorithms, and applications is accessible online [1].

The thesis of this paper is that the unfolding method is applicable to any model of concurrency for which a notion of 'events occurring independently from each other' can be defined, and not only to Petri nets—as is often assumed. We provide evidence in favour of this thesis by applying the method to *synchronous products of labelled transition systems*. In this model, introduced by Arnold in [2], a system consists of a tuple of communicating sequential components. The communication discipline, formalised by means of so-called synchronisation vectors, is very general, and contains as special cases the communication mechanisms of process algebras like CCS and CSP.

Readers acquainted with both Arnold's and the Petri net model will probably think that our task is not very difficult, and they are right. It is indeed straightforward to give synchronous products of transition systems a Petri net semantics, and then apply the usual machinery. But we go a bit further: We show that the

* Work partially supported by the Teilprojekt A3 SAM of the Sonderforschungsbereich 342 "Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen".

additional structure of Arnold's model with respect to Petri nets—the fact that we are given a decomposition of the system into sequential components—can be used to simplify the unfolding method. More precisely, in a former paper by Vogler and the authors [4], we showed that the key to an efficient algorithm for the construction of a complete finite prefix is to find a mathematical object called a *total adequate order*, and provided such an order for systems modelled by Petri nets[1]. In this paper we present a new total adequate order for synchronous products of labelled transition systems. The proof of adequacy for this new order is simpler than the proof of [4].

In a second part of the paper we describe an efficient implementation of the algorithm , and compare it with the algorithm of [4] on a set of benchmarks.

Very recently, further evidence for the wide applicability of unfoldings has been provided by Langerak and Brinksma in [5]. Independently from us, they have applied the unfolding technique to a CSP-like process algebra, a model even further away from Petri nets than ours. A brief discussion of the relation to our work can be found in the conclusions.

The paper is organised as follows. Section 2 introduces synchronous products of transition systems following [2], and Section 3 gives them a partial order semantics based on unfoldings. Section 4 describes an algorithm to construct a complete finite prefix. Section 5 discusses how to efficiently implement it. Section 6 discusses the performance of the new algorithm.

2 Synchronous Products of Transition Systems

In this section we introduce Arnold's model and its standard interleaving semantics. Notations follow [2] with very few minor changes.

2.1 Labelled Transition Systems

A *labelled transition system* is a tuple $\mathcal{A} = \langle S, T, \alpha, \beta, \lambda \rangle$, where S is a set of *states*, T is a set of *transitions*, $\alpha, \beta : T \to S$ are the *source* and *target* mappings, and $\lambda : T \to A$ is a labelling mapping assigning to each transition a letter from an alphabet A. We assume that A contains a special label ϵ, and that for each state $s \in S$ there is a transition ϵ_s such that $\alpha(\epsilon_s) = s = \beta(\epsilon_s)$, and $\lambda(\epsilon_s) = \epsilon$. Moreover, no other transitions are labelled by ϵ. Transitions labelled by ϵ are called *idle* transitions in the sequel.

We use a graphical representation for labelled transition systems. States are represented by circles, and a transition t with $\alpha(t) = s$, $\beta(t) = s'$, and $\lambda(t) = a$ is represented by an arrow leading from s to s' labelled by $t : a$. Idle transitions are not represented. Figure 1 shows two labelled transition systems.

[1] More exactly, systems modelled by 1-safe Petri nets, i.e., Petri nets whose places can hold at most one token.

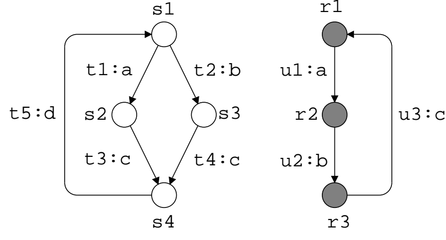

Fig. 1. Two labelled transition systems

2.2 Synchronous Products

Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be labelled transition systems, where $\mathcal{A}_i = \langle S_i, T_i, \alpha_i, \beta_i, \lambda_i \rangle$, and λ_i labels each transition of T_i with an element of an alphabet A_i. We assume for convenience that the sets S_i and T_i are pairwise disjoint. A subset I of $(A_1 \times \ldots \times A_n) \setminus (\epsilon, \ldots, \epsilon)$ is called a *synchronisation constraint*, and the elements of I are called *synchronisation vectors*. Loosely speaking, these vectors indicate which transitions of $\mathcal{A}_1, \ldots, \mathcal{A}_n$ must synchronise. The tuple $\mathbf{A} = \langle \mathcal{A}_1, \ldots, \mathcal{A}_n, I \rangle$ is called the *synchronous product* of the $\mathcal{A}_i$ under I.

As running example we use $\mathbf{A} = \langle \mathcal{A}_1, \mathcal{A}_2, I \rangle$, where $\mathcal{A}_1, \mathcal{A}_2$ are the two labelled transition systems of Figure 1, and I contains the following synchronisation vectors:

$$\langle a, \epsilon \rangle, \langle b, \epsilon \rangle, \langle d, \epsilon \rangle, \langle \epsilon, a \rangle, \langle \epsilon, c \rangle, \langle c, b \rangle$$

I.e., c-labelled transitions of $\mathcal{A}_1$ must synchronise with b-labelled transitions of $\mathcal{A}_2$. The other transitions do not synchronise.

The interleaving semantics of $\mathbf{A}$ is the labelled transition system $A_{int} = \langle S, T, \alpha, \beta, \lambda \rangle$, where $\lambda \colon T \to I$, and

$$\begin{aligned} S &= S_1 \times \ldots \times S_n \\ T &= \{\langle t_1, \ldots, t_n \rangle \mid \langle \lambda_1(t_1), \ldots, \lambda_n(t_n) \rangle \in I\} \\ \alpha(\langle t_1, \ldots, t_n \rangle) &= \langle \alpha_1(t_1), \ldots, \alpha_n(t_n) \rangle \\ \beta(\langle t_1, \ldots, t_n \rangle) &= \langle \beta_1(t_1), \ldots, \beta_n(t_n) \rangle \\ \lambda(\langle t_1, \ldots, t_n \rangle) &= \langle \lambda_1(t_1), \ldots, \lambda_n(t_n) \rangle \end{aligned}$$

The elements of S and T are called *global states* and global transitions, respectively.

If each of the $\mathcal{A}_i$ has a distinguished initial state is_i, then the initial state of $\mathbf{A}$ is the tuple $\mathbf{is} = \langle is_1, \ldots, is_n \rangle$, and $\mathbf{A}$ with $\mathbf{is}$ as initial state is denoted by $\langle \mathbf{A}, \mathbf{is} \rangle$. The set of *reachable* global states is then the set of global states reachable from $\mathbf{is}$. For our running example we take $\mathbf{is} = \langle s_1, r_1 \rangle$.

We introduce a notation that will help us to later define the unfolding of $\mathbf{A}$. Given a global transition $\mathbf{t} = \langle t_1, \ldots, t_n \rangle$ of $\mathbf{A}$, we define

$$\begin{aligned} {}^\bullet\mathbf{t} &= \{\alpha_i(t_i) \mid 1 \le i \le n \text{ and } \lambda_i(t_i) \ne \epsilon\} \\ \mathbf{t}^\bullet &= \{\beta_i(t_i) \mid 1 \le i \le n \text{ and } \lambda_i(t_i) \ne \epsilon\} \end{aligned}$$

Loosely speaking, ${}^\bullet\mathbf{t}$ contains the sources of the non-idle transitions of $\mathbf{t}$, and $\mathbf{t}^\bullet$ their targets.

3 Unfolding of a Synchronous Product

In [2], synchronous products are only given an interleaving semantics. In this section we give them a partial order semantics based on the notion of unfolding of a synchronous product, and show its compatibility with the interleaving semantics. We introduce a number of standard notions about Petri nets, but sometimes our definitions are not completely formalised. The reader interested in rigorous definitions is referred to [4].

3.1 Petri nets

As usual, a *net* consists of a set of *places*, graphically represented by circles, a set of *transitions*, graphically represented as boxes, and a flow relation assigning to each place (transition) a set of input and a set of output transitions (places). The flow relation is graphically represented by arrows leading from places to transitions and from transitions to places. In order to avoid confusions between the transitions of a transition system and the transitions of a Petri net, we call the latter *events* in the sequel. Places and events are called *nodes*; given a node x, the set of input and output nodes of x is denoted by ${}^\bullet x$ and $x^\bullet$, respectively. A place of a net can hold tokens, and a mapping assigning to each place a number of tokens is called a *marking*. If, at a given marking, all the input places of an event hold at least one token, then the event can *occur*, which leads to a new marking obtained by removing one token from each input place and adding one token to each output place. An *occurrence sequence* is a sequence of events that can occur in the order specified by the sequence.

A synchronous product can be associated a Petri net as follows: Take a place for each state of each component, and an event for each global transition; add an arc from s to $\mathbf{t}$ if $s \in {}^\bullet\mathbf{t}$, and from $\mathbf{t}$ to s if $s \in \mathbf{t}^\bullet$; put a token in the initial state of each component, and no tokens elsewhere. The unfolding of a synchronous product can be defined as the unfolding of its associated Petri net, but in the rest of the section we give a direct definition.

3.2 Ocurrence nets

Given two nodes x and y of a net, we say that x is *causally related* to y, denoted by $x \leq y$, if there is a (possibly empty) path of arrows from x to y. We say that x and y are in *conflict*, denoted by $x \# y$, if there is a place z, different from x and y, from which one can reach x and y, exiting x by different arrows. Finally, we say that x and y are *concurrent*, denoted by $x \, co \, y$, if neither $x \leq y$ nor $y \leq x$ nor $x \# y$ hold. *Occurrence nets* are those satisfying the following three properties:

- the net, seen as a graph, has no cycles;
- every place has at most one input event;
- no node is in self-conflict, i.e., $x \# x$ holds for no x.

The nets of Figure 2 and Figure 3 are occurrence nets.

Occurrence nets can be infinite. We restrict ourselves to those in which every event has at least one input place, and in which the arrows cannot be followed backward infinitely from any point (this is called *well-foundedness*). It follows that by following the arrows backward we eventually reach a place without predecessors. These are the *minimal places* of the net.

We associate to an occurrence net a default *initial marking*, in which the minimal places carry exactly one token, and the other places no tokens. It is easy to see that all the markings reachable from the initial marking also put at most one token on a place. Therefore, we represent reachable markings as sets of places.

3.3 Branching processes

Given a synchronous product of transition systems, we associate to it a set of *labelled* occurrence nets, called the *branching processes* of $\mathbf{A}$. The places[2] of these nets are labelled with states of the components of $\mathbf{A}$, and their events are labelled with global transitions. The places and events of the branching processes are all taken from two sets $\mathcal{P}$ and $\mathcal{E}$, inductively defined as follows:

- $\bot \in \mathcal{E}$, where $\bot$ is a special symbol;
- if $e \in \mathcal{E}$, then $(s, e) \in \mathcal{P}$ for every $s \in S_1 \cup \ldots \cup S_n$;
- if $X \subseteq \mathcal{P}$, then $(\mathbf{t}, X) \in \mathcal{E}$ for every $\mathbf{t} \in T$.

In our definition of branching process (see below) we make consistent use of these names: The label of a place (s, e) is s, and its unique input event is e. Places $(s, \bot)$ are those having no input event, i.e., the special symbol $\bot$ is used for the minimal places of the occurrence net. Similarly, the label of an event $(\mathbf{t}, X)$ is $\mathbf{t}$, and its set of input places is X. The advantage of this scheme is that a branching process is completely determined by its sets of places and events. In the sequel, we make use of this and represent a branching process as a pair (P, E).

Definition 1. *The set of* finite branching processes of $\langle \mathbf{A}, \mathbf{is} \rangle$, *where* $\mathbf{is} = \langle is_1, \ldots, is_n \rangle$*, is inductively defined as follows:*

- $(\{(is_1, \bot), \ldots, (is_n, \bot)\}, \emptyset)$ *is a branching process of* $\langle \mathbf{A}, \mathbf{is} \rangle$.
- *If* (P, E) *is a branching process,* $\mathbf{t}$ *is a global transition, and* $X \subseteq P$ *is a co-set labelled by* ${}^\bullet\mathbf{t}$*, then*

$$(\, P \cup \{(s, e) | s \in \mathbf{t}^\bullet\} \, , \; E \cup \{e\} \,)$$

is also a branching process of $\langle \mathbf{A}, \mathbf{is} \rangle$*, where* $e = (\mathbf{t}, X)$*. If* $e \notin E$*, then* e *is called a* possible extension *of* (P, E).

We denote the set of possible extensions of a branching process BP by PE(BP).

---

[2] In some papers (including [4]), the name *conditions* is used instead of places.

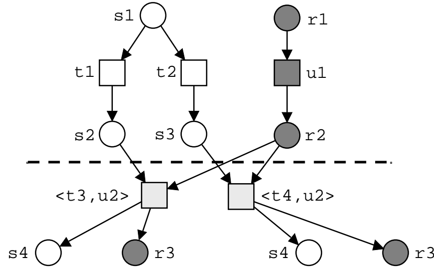

Fig. 2. A branching process of $\langle \mathbf{A}, \mathbf{is} \rangle$

A place of the form $(s, \bot)$ or (s, e) such that $s \in S_i$ is called an *i-place*. An event of the form $(\mathbf{t}, X)$ such that $\mathbf{t}(i)$ is not an idle transition is called an *i-event*. Observe that an event can be both an i-event and a j-event for $i \neq j$ (in this case we say that $\mathcal{A}_i$ and $\mathcal{A}_j$ *synchronize* in e), but a place cannot, since the states of the different components are disjoint by assumption.

Figure 2 shows a finite branching process of our running example (above the dashed line), together with its two possible extensions (below that line). 1-nodes are white, 2-nodes are dark grey, and events that are both 1- and 2 events are light grey. The labels of events have been simplified for clarity: We write t instead of $\langle t, \epsilon \rangle$, and u instead of $\langle \epsilon, u \rangle$.

The set of branching processes of $\langle \mathbf{A}, \mathbf{is} \rangle$ is obtained by declaring that the union of any finite or infinite set of branching processes is also a branching process, where union of branching processes is defined componentwise on places and events. Since branching processes are closed under union, there is a unique maximal branching process. We call it the *unfolding* of $\langle \mathbf{A}, \mathbf{is} \rangle$. The unfolding of our running example is an infinite occurrence net. Figure 3 shows an initial part. Events and places have been assigned identificators that will be used in the examples.

The following Proposition is easy to prove by structural induction on branching processes:

Proposition 1. *Two i-nodes of a branching process are either causally related or in conflict.*

For instance, in Figure 3 all white and light grey nodes are causally related or in conflict.

3.4 Configurations and cuts

For our purposes, the most interesting property of occurrence nets is that their sets of occurrence sequences and reachable markings can be easily characterised in graph-theoretic terms using the notions of configuration and cut.

Definition 2. *A* configuration *of an occurrence net is a set of events C satisfying the two following properties: C is causally closed, i.e., if $e \in C$ and $e' < e$ then $e' \in C$, and C is conflict-free, i.e., no two events of C are in conflict.*

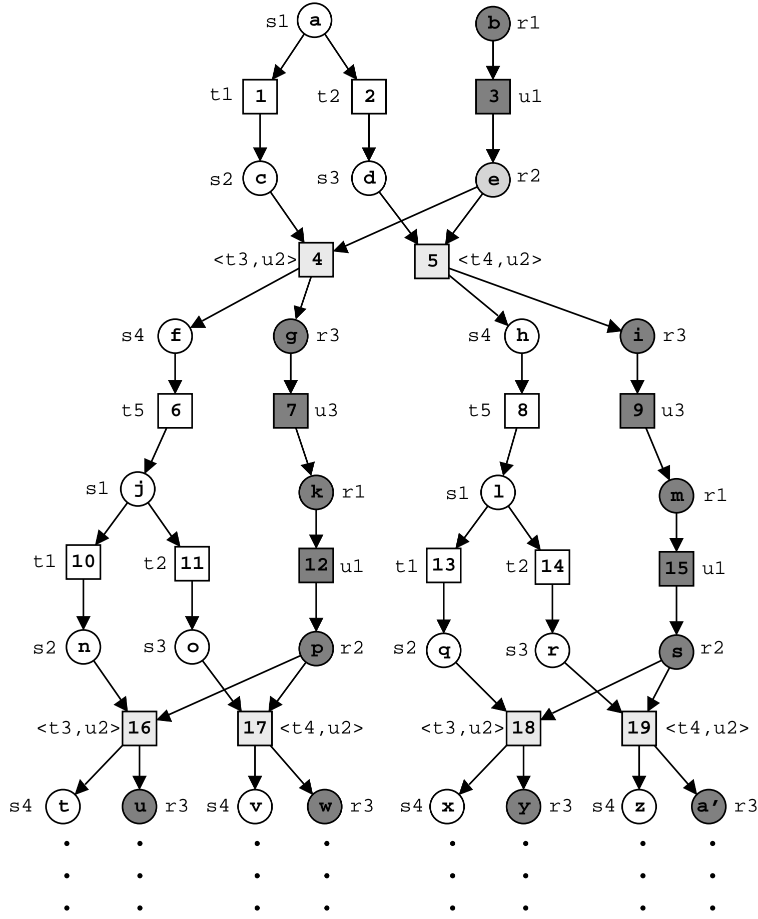

Fig. 3. The unfolding of $\langle \mathbf{A}, \mathbf{is} \rangle$

In Figure 3, $\{1, 3, 4, 6\}$ is a configuration, and $\{1, 4\}$ (not causally closed) or $\{1, 2\}$ (not conflict-free) are not.

It is easy to prove that a set of events is a configuration if and only if there is an occurrence sequence of the net (from the default initial marking) containing each event from the set exactly once, and no further events. This occurrence sequence is not necessarily unique. For instance, for the configuration $\{1, 3, 4, 6\}$ there are two occurrence sequences like $1\,3\,4\,6$ or $3\,1\,4\,6$. However, all occurrence sequences corresponding to the same configuration lead to the same reachable marking. For example, the two sequences above lead to the marking $\{j, g\}$.

Definition 3. *A* cut *is a set of places* $\mathbf{c}$ *satisfying the two following properties:* $\mathbf{c}$ *is a co-set, i.e., any two elements of* $\mathbf{c}$ *are concurrent, and* $\mathbf{c}$ *is maximal, i.e., it is not properly included in any other co-set.*

It is easy to prove that the reachable markings of an occurrence net coincide with its cuts. We can assign to a configuration C the marking reached by any of the occurrence sequences mentioned above. This marking is a cut, and it is easy to prove that it is equal to $(Min \cup {}^\bullet C) \setminus C^\bullet$, where Min denotes the set of minimal places of the branching process.

The following Proposition can also be easily proved by structural induction on branching processes.

Proposition 2. *A cut* $\mathbf{c}$ *of a branching process contains exactly one i-place for each component* $\mathcal{A}_i$.

This result allows us to use the notation $\mathbf{c} = \langle p_1, \ldots, p_n \rangle$ for cuts. Since the place p_i is labelled by some state $s_i \in S_i$, the tuple $\langle s_1, \ldots, s_n \rangle$ is a reachable global state of $\langle \mathbf{A}, \mathbf{is} \rangle$. The global state corresponding to the cut of a configuration C is denoted by $GState(C)$.

We take as partial order semantics of $\langle \mathbf{A}, \mathbf{is} \rangle$ its unfolding. The relationship between the interleaving and partial order semantics of $\langle \mathbf{A}, \mathbf{is} \rangle$ is given by the following result:

Theorem 1. *Let* $\langle \mathbf{A}, \mathbf{is} \rangle$ *be a synchronous product of transition systems.*

(a) Let C *be a configuration of a branching process of* $\langle \mathbf{A}, \mathbf{is} \rangle$. *There is a state* $\mathbf{s}$ *of* $\mathcal{A}_{int}$, *reachable from* $\mathbf{is}$, *such that: (1)* $\mathbf{s} = GState(C)$, *and (2) for every configuration* $C \cup \{e\}$ $(e \notin C)$ *there is a transition* $\mathbf{t}$ *of* $\mathcal{A}_{int}$ *such that* $\alpha(\mathbf{t}) = GState(C)$ *and* $\beta(\mathbf{t}) = GState(C \cup \{e\})$.

(b) Let $\mathbf{s}$ *be a state of* $\mathcal{A}_{int}$, *reachable from* $\mathbf{is}$. *There is a configuration* C *of the unfolding of* $\langle \mathbf{A}, \mathbf{is} \rangle$ *such that: (1)* $GState(C) = \mathbf{s}$, *and (2) for every transition* $\mathbf{t}$ *of* $\mathcal{A}_{int}$ *such that* $\alpha(\mathbf{t}) = \mathbf{s}$ *and* $\beta(\mathbf{t}) = s'$ *there exists a configuration* $C \cup \{e\}$ $(e \notin C)$ *such that* e *is labelled by* $\mathbf{t}$, *and* $GState(C \cup \{e\}) = \mathbf{s}'$.

Informally, (a) means that the information a branching process has about $\mathcal{A}_{int}$ is correct, while (b) means that the unfolding has complete information about $\mathcal{A}_{int}$ (actually, the unfolding also contains "true concurrency" information).

4 Constructing a complete finite prefix

We say that a branching process of $\langle \mathbf{A}, \mathbf{is} \rangle$ is *complete* if it contains complete information about $\mathcal{A}_{int}$, i.e., if condition (b) of Theorem 1, which is always fulfilled by the unfolding, also holds for it.[3] The important fact is that *finite* complete prefixes exist, the main reason being that the number of global states of $\langle \mathbf{A}, \mathbf{is} \rangle$ is finite. For instance, the prefix of Figure 3 containing the places $\{a, \ldots, k, n, o, p\}$ and the events $\{1, \ldots, 7, 10, 11, 12\}$ can be shown to be a complete prefix.

[3] In fact, it is easy to see that a complete prefix contains as much information as the unfolding itself, in the sense that given a complete prefix there is a unique unfolding containing it.

In [4] an algorithm is presented for the construction of a complete finite prefix, which improves on a previous construction presented in [6]. The algorithm makes use of a so-called *adequate order* on the configurations of the unfolding. Different adequate orders lead to different versions of the algorithm, and also to different complete prefixes. Total adequate orders are particularly nice, since they lead to complete prefixes which, loosely speaking, are guaranteed not to be larger than the transition system $\mathcal{A}_{int}$[4]. In [4] a total adequate order for the unfoldings of Petri nets is presented. In this section we recall the algorithm of [4], and then present a total adequate order for the unfoldings of synchronous products of transition systems. The additional structure of a synchronous product with respect to a Petri net leads to a simpler order, with a simpler proof of adequacy.

4.1 The algorithm

Given a configuration C of the unfolding, we denote by $C \oplus E$ the set $C \cup E$, under the condition that $C \cup E$ is a configuration satisfying $C \cap E = \emptyset$. We say that $C \oplus E$ is an *extension* of C, and that E is a *suffix* of $C \oplus E$. Obviously, if $C \subseteq C'$ then there is a suffix E of C' such that $C \oplus E = C'$.

Now, let C_1 and C_2 be two finite configurations leading to the same global state, i.e. $GState(C_1) = \mathbf{s} = GState(C_2)$. The 'continuations' of the unfolding from the cuts corresponding to C_1 and C_2 (the nodes lying below these cuts) are isomorphic (see [4] for a more formal description). For example, in Figure 3 the configurations $\{1, 3, 4\}$ and $\{2, 3, 5\}$ lead to the cuts $\langle f, g\rangle$ and $\langle h, i\rangle$, which correspond to the global state $\langle s_4, r_3\rangle$. Loosely speaking, the continuations from these cuts contain the nodes below f, g and h, i, respectively (f, g and h, i included). This isomorphism, say I, induces a mapping from the extensions of C_1 onto the extensions of C_2, which maps $C_1 \oplus E$ onto $C_2 \oplus I(E)$. For example, $\{1, 3, 4, 7, 12\}$ is mapped onto $\{2, 3, 5, 9, 15\}$.

The intuitive idea behind the algorithm is to avoid computing isomorphic continuations, since one representative suffices. However, a correct formalisation is not easily achieved. It requires the following three basic notions:

Definition 4. *A partial order $\prec$ on the finite configurations of the unfolding is* adequate *if:*

- *it is well-founded,*
- *it refines the inclusion order, i.e. $C_1 \subset C_2$ implies $C_1 \prec C_2$, and*
- *it is preserved by finite extensions, i.e. if $C_1 \prec C_2$ and $GState(C_1) = GState(C_2)$, then the isomorphism I above satisfies $C_1 \oplus E \prec C_2 \oplus I(E)$ for all finite extensions $C_1 \oplus E$ of C_1.*

Definition 5. *The local configuration $[e]$ associated to an event e of a branching process is the set of events e' such that $e' \leq e$.*[5]

[4] For a more precise statement see [4].
[5] It is immediate to prove that $[e]$ is a configuration.

Definition 6. *Let $\prec$ be an adequate order on the configurations of the unfolding, and let BP be a branching process containing an event e. The event e is a* cut-off event *of BP (with respect to $\prec$) if BP contains a local configuration $[e']$ such that $GState([e]) = GState([e'])$, and $[e'] \prec [e]$.*

The algorithm is in fact a family of algorithms: each adequate order $\prec$ leads to a different member of the family. It computes a branching process, and whenever it identifies a cut-off event it takes care of not extending the process behind it.

input: a synchronous product $\langle \mathbf{A}, \mathbf{is} \rangle$, where $\mathbf{is} = \langle is_1, \ldots, is_n \rangle$.
output: a complete finite prefix of the unfolding of $\langle \mathbf{A}, \mathbf{is} \rangle$.
begin
 $bp := (\{(is_1, \bot), \ldots, (is_n, \bot)\}, \emptyset)$;
 $pe := PE(bp)$;
 $\textit{cut-off} := \emptyset$;
 while $pe \neq \emptyset$ **do**
  choose $e = (\mathbf{t}, X)$ in pe such that $[e]$ is minimal with respect to $\prec$;
  if $[e] \cap \textit{cut-off} = \emptyset$ **then**
   extend bp with the event e and with a place (s, e)
    for every output place s of $\mathbf{t}$;
   $pe := PE(bp)$;
   if $GState([e]) = GState([e'])$ for some event e' of bp **then**
    $\textit{cut-off} := \textit{cut-off} \cup \{e\}$
   endif
  else $pe := pe \setminus \{e\}$
  endif
 endwhile;
 return bp
end

One of the main results of [4] states that this algorithm is correct if $\prec$ is an adequate order. The order $\prec$ need not be total, but, loosely speaking, total orders lead to more cut-off events, and so to smaller prefixes. In fact, totality is a sufficient condition for the output of the algorithm to be at most as large as the interleaving semantics $\mathcal{A}_{int}$. Weaker conditions achieve the same effect (the order need only be total among configurations with the same associated global state, a fact exploited in [5]), but we do not need them here.

5 Adequate orders for the unfolding of a synchronous product

In this section we introduce a total adequate order on the configurations of the unfolding of a synchronous product. The order is simpler to define and to prove adequate than the order introduced in [4] for systems modelled by Petri nets.

5.1 Local views

Our adequate order is based on the notion of *local view* of a configuration. Given a finite configuration C, we define its *projection* $C|_i$ *onto* $\mathcal{A}_i$ as its set of i-events. If we take $C = \{2, 3, 5, 8, 9, 13\}$ in Figure 3, then we have $C|_1 = \{2, 5, 8, 13\}$ and $C|_2 = \{3, 5, 9\}$. The events of $C|_i$ are totally ordered by the causal relation $<$. This is so because i-events are either causally related or in conflict (Proposition 1), and the events of $C|_i$ are not in conflict because they belong to a configuration. We define:

Definition 7. *Let C be a configuration, and let $e_1 < e_2 < \ldots < e_{k_i}$ be the result of ordering $C|_i$ with respect to $<$. The i-*view *of a configuration C, denoted by $V_i(C)$, is the sequence $\mathbf{t}_1\mathbf{t}_2 \ldots \mathbf{t}_{k_i}$, where $\mathbf{t}_j$ is the global transition labelling the event e_j. We denote by $\mathbf{V}(C) = \langle V_1(C), \ldots, V_n(C)\rangle$ the n-tuple of local views of a configuration.*

Intuitively, $V_i(C)$ is the history of the computation as seen by the i-th component. In our example we have $2 < 5 < 8 < 13$ for $C|_1$ and $3 < 5 < 9$ for $C|_2$. Furthermore, $V_1(C) = t_2\langle t_4, u_2\rangle t_5 t_1$ and $V_2(C) = u_1\langle t_4, u_2\rangle u_3$.

The definition of local view can be extended without problems to suffixes of configurations, for instance to the set $\{8, 9\}$: We have then $V_1(\{8, 9\}) = \langle t_5, \epsilon\rangle$ and $V_2(\{8, 9\}) = \langle \epsilon, u_3\rangle$. In particular, for an event $e = (\mathbf{t}, X)$ we have that $V_i(\{e\})$ is the empty sequence if $\mathbf{t}(i)$ is an idle transition, and $V_i(\{e\}) = \mathbf{t}$ otherwise.

The following result will be crucial:

Theorem 2. *The mapping* $\mathbf{V}$ *is injective.*

Proof. Let $C_1 = C_2$ be two configurations such that $\mathbf{V}(C_1) = \mathbf{V}(C_2)$. We prove $C_1 = C_2$ by showing $C_1 = C$ and $C_2 = C$, where $C = C_1 \cap C_2$. By symmetry it suffices to prove $C_1 = C$. We proceed by contradiction.

Assume $C \neq C_1$. Then C can be extended by an event $e_1 \in C_1 \setminus C$. We prove $e_1 \in C_2$, a contradiction to $C = C_1 \cap C_2$. Let $e_1 = (\mathbf{t}, X_1)$, where $\mathbf{t} = \langle t_1, \ldots, t_n\rangle$. Since $\mathbf{t} \neq (\epsilon, \ldots, \epsilon)$ by the definition of global transition, some component of $\mathbf{t}$, say t_i, satisfies $\lambda_i(t_i) \neq \epsilon$. By the definition of local view, $V_i(C) \cdot \mathbf{t}$ is a prefix of $V_i(C_1)$, and, since $\mathbf{V}(C_1) = \mathbf{V}(C_2)$ holds by assumption, also a prefix of $V_i(C_2)$. So C can be extended by an event $e_2 \in C_2$ such that $e_2 = (\mathbf{t}, X_2)$ for some *co*-set X_2. We prove:

- X_1 and X_2 are both labelled by $^\bullet\mathbf{t}$. Follows immediately from $e_1 = (\mathbf{t}, X_1)$ and $e_2 = (\mathbf{t}, X_2)$.
- Each place of $X_1 \cup X_2$ carries a different label. Since both e_1 and e_2 extend the same configuration C, we have that $X_1 \cup X_2$ is a *co*-set. Since every *co*-set can be extended to a cut, we can apply Proposition 4.

It follows $X_1 = X_2$, which implies $e_1 = e_2$. So $e_1 \in C_2$, and we are done.

In words, Theorem 2 states that a configuration is characterised by its tuple of local views. If we let $\mathbf{T}^*$ be the set of n-tuples whose elements are sequences of global transitions, i.e., $\mathbf{T}^* = (T^*)^n$, then a tuple of local views is an element of $\mathbf{T}^*$. By Theorem 2, an order $\prec$ on $\mathbf{T}^*$ induces an order on configurations:

$$C_1 \prec_C C_2 \text{ if and only if } \mathbf{V}(C_1) \prec \mathbf{V}(C_2)$$

Moreover, if $\prec$ is total, then $\prec_C$ is total.

5.2 From orders on local views to adequate orders

We identify sufficient conditions for an order $\prec$ on $\mathbf{T}^*$ to induce an *adequate* total order on configurations. We need to introduce some definitions. The concatenation of two elements $\sigma, \tau \in \mathbf{T}^*$ is defined componentwise, and denoted by $\sigma \cdot \tau$. The partial order $\sqsubseteq$ on $\mathbf{T}^*$ is defined as follows: $\sigma \sqsubseteq \tau$ if there exists σ' such that $\tau = \sigma \cdot \sigma'$. In other words, $\sigma \sqsubseteq \tau$ if each component of σ is a prefix of the corresponding component of τ.

We start with the following two observations, which follow easily from the definitions.

Proposition 3. *(1) If $C_1 \subseteq C_2$ then $\mathbf{V}(C_1) \sqsubseteq \mathbf{V}(C_2)$.*
(2) $\mathbf{V}(C \oplus E) = \mathbf{V}(C) \cdot \mathbf{V}(E)$.

Let us illustrate this result with configurations from Figure 3. Let $C_1 = \{2, 3, 5, 8\}$ and $C_2 = \{2, 3, 5, 8, 9, 13\}$. We have

$$\begin{aligned} \mathbf{V}(C_1) &= \{t_2\langle t_4, u_2\rangle t_5, u_1\langle t_4, u_2\rangle\} \\ \mathbf{V}(C_2) &= \{t_2\langle t_4, u_2\rangle t_5 t_1, u_1\langle t_4, u_2\rangle u_3\} \end{aligned}$$

Furthermore, we have $C_2 = C_1 \oplus E$, where $E = \{9, 13\}$, and $\mathbf{V}(E) = \{t_1, u_3\}$, and indeed $\mathbf{V}(C_2) = \mathbf{V}(C_1) \cdot \mathbf{V}(E)$.

We can now obtain sufficient conditions for the induced order $\prec_C$ to be adequate and total:

Lemma 1. *Let $\prec$ be an order on $\mathbf{T}^*$ satisfying the following conditions:*
(1) $\prec$ is well-founded;
(2) $\prec$ refines $\sqsubseteq$, i.e. $\sigma \sqsubseteq \tau$ implies $\sigma \prec \tau$;
(3) $\prec$ is preserved by concatenation, i.e., if $\sigma \prec \tau$ then $\sigma \cdot \sigma' \prec \tau \cdot \sigma'$ for every $\sigma' \in \mathbf{T}^*$;
(4) $\prec$ is a total order.
Then the induced order $\prec_C$ is a total adequate order.

Proof. We prove that $\prec_C$ satisfies the properties of a total adequate order:

(a) $\prec_C$ is well-founded. $C_1 \succ_C C_2 \succ_C \ldots$ implies $\mathbf{V}(C_1) \succ \mathbf{V}(C_2) \succ \ldots$, contradicting the well-foundedness of $\prec$.
(b) If $C_1 \subseteq C_2$ then $C_1 \prec_C C_2$. By Proposition 3(1), $\mathbf{V}(C_1) \sqsubseteq \mathbf{V}(C_2)$. By (2), $\mathbf{V}(C_1) \prec \mathbf{V}(C_2)$. By the definition of $\prec_C$, $C_1 \prec_C C_2$.
(c) If $C_1 \prec_C C_2$ then $C_1 \oplus E \prec_C C_2 \oplus E$. If $C_1 \prec_C C_2$ then $\mathbf{V}(C_1) \prec \mathbf{V}(C_2)$. By (3), $\mathbf{V}(C_1) \cdot \mathbf{V}(E) \prec \mathbf{V}(C_2) \cdot \mathbf{V}(E)$. By Proposition 3(2), $\mathbf{V}(C_1 \oplus E) \prec \mathbf{V}(C_2 \oplus E)$. By the definition of $\prec_C$, $C_1 \oplus E \prec C_2 \oplus E$.
(d) $\prec_C$ is total. Immediate from (4) and the definition of $\prec_C$.

5.3 Orders on $\mathbf{T}^*$ inducing adequate orders

We describe in this section two total orders on $\mathbf{T}^*$ satisfying conditions (1)–(4) of Lemma 1. We start with an arbitrary total order $\prec_T$ on T, and use the following three auxiliary orders on T^*:

- the *size* order: σ is smaller than τ if $|\sigma| < |\tau|$;
- the *lexicographic* order: σ is smaller than τ if σ is lexicographically smaller than τ with respect to $\prec_T$.
- the *silex* (size-lexicographic) order: σ is smaller than τ if $|\sigma| < |\tau|$ or if $|\sigma| = |\tau|$ and σ is lexicographically smaller than τ.

Let us first consider the case $n = 1$, i.e, $\mathbf{A}$ contains only one component. We have then $\mathbf{T}^* = T^*$, i.e., we look for an order on sequences of global transitions satisfying (1)–(4). It is immediate to see that the silex order does the job: the order $\sqsubseteq$ is in this case the prefix order on sequences, and the concatenation operation is just the ordinary concatenation of sequences.

The silex order can be extended to an arbitrary number n of components in two different ways:

Definition 8. *Let σ, τ be elements of $\mathbf{T}^*$. We say $\sigma \prec_1 \tau$ if there is an index $1 \leq i \leq n$ such that $\sigma(j) = \tau(j)$ for all $1 \leq j < i$, and $\sigma(i)$ is smaller than $\tau(i)$ with respect to the silex order on sequences. We say $\sigma \prec_2 \tau$ if*

(a) there is an index $1 \leq i \leq n$ such that $|\sigma(j)| = |\tau(j)|$ for all $1 \leq j < i$, and $|\sigma(i)| < |\tau(i)|$, or

(b) $|\sigma(i)| = |\tau(i)|$ for all $1 \leq i \leq n$, and there is an index i such that $\sigma(j) = \tau(j)$ for all $1 \leq j < i$ and $\sigma(i)$ is lexicographically smaller than $\tau(i)$.

It is only a small exercise to prove that $\prec_1$ and $\prec_2$ satisfy conditions (1)–(4):

Theorem 3. *The orders $\prec_1$ and $\prec_2$ satisfy conditions (1)–(4) of Lemma 1. Therefore, they induce total adequate orders on configurations.*

Proof. Let us prove condition (3) for the order $\prec_2$, the others being similar or simpler. Assume $\sigma \prec_2 \tau$. We prove $\sigma \cdot \sigma \prec_2 \tau \cdot \sigma$. Let $\sigma(i)$ be the first component of σ such that $\sigma(i)$ is smaller than $\tau(i)$ with respect to the silex order. Consider two cases:

- There is an index $1 \leq i \leq n$ such that $|\sigma(j)| = |\tau(j)|$ for all $1 \leq j < i$ and $|\sigma(i)| < |\tau(i)|$. Then $|\sigma(j)\sigma'(j)| = |\tau(j)\sigma'(j)|$ for all $1 \leq j < i$ and $|\sigma(i)\sigma'(i)| < |\tau(i)\sigma'(i)|$. Hence $\sigma \cdot \sigma \prec_2 \tau \cdot \sigma$.
- $|\sigma(i)| = |\tau(i)|$ for all $1 \leq i \leq n$, and there is an index i such that $\sigma(j) = \tau(j)$ for all $1 \leq j < i$, and $\sigma(i)$ is lexicographically smaller than $\tau(i)$. Then $|\sigma(i)\sigma'(i)| = |\tau(i)\sigma'(i)|$ for all $1 \leq i \leq n$, and there is an index i such that $\sigma(j)\sigma'(j) = \tau(j)\sigma'(j)$ for all $1 \leq j < i$ and $\sigma(i)\sigma'(i)$ is lexicographically smaller than $\tau(i)\sigma'(i)$. Hence $\sigma \cdot \sigma \prec_2 \tau \cdot \sigma$.

This concludes the proof of adequacy of the two orders $\prec_1$ and $\prec_2$. The proof consists of Theorem 2, Lemma 1, and Theorem 3. The latter two have very simple proofs, only Theorem 2 requires a bit of ingenuity.

Which of the two orders is more suitable for an implementation is a question of efficiency, and is discussed—together with other implementation points—in the next section.

6 Efficient implementation of the complete finite prefix algorithm

The algorithm presented in Section 4.1 is hopefully easy to understand. However, it is still far too abstract. It leaves the choice of the order $\prec$ open, and it does not explain how to compute the functions *PE* and *GState*, nor how to compute a minimal event with respect to $\prec$. In the algorithm of [4] the computation of the functions and the minimal event involved expensive forward and backward global searches in branching processes. The additional structure of synchronous products allows to compute *GState* and minimal events using new procedures, described in Sections 6.2 and 6.1, respectively. In Section 6.3 we also describe how to speed-up the computation of *PE*; however, in this case the improvement does not exploit the structure of synchronous products, and can be used for Petri net systems as well.

In the sequel, the abstract algorithm of the last section is called 'the algorithm'. The concrete algorithm using the procedures just mentioned is called 'our implementation'.

6.1 Computing a minimal event

In order to determine the minimal event, our implementation maintains a queue of possible extensions sorted according to $\prec_C$. So we need a procedure to decide for two given configurations $[e_1], [e_2]$ whether $[e_1] \prec_C [e_2]$ or $[e_2] \prec_C [e_1]$. For both $\prec = \prec_1$ and $\prec = \prec_2$ we face a trade-off between time and space. The fastest procedure is to attach to each event e in the queue the whole vector $\mathbf{V}([e])$, which leads to a high memory overhead. The most economic procedure in memory terms is to recompute $\mathbf{V}([e])$ whenever it is needed by means of a backward search, a much slower solution. In our implementation we adopt an intermediate solution: We attach to each event e in the queue the integer vector $\langle |V_1([e])|, \ldots, |V_n([e])| \rangle$.

Once this design choice has been made, the order $\prec_2$ becomes superior to $\prec_1$. With $\prec = \prec_2$, the vectors $\mathbf{V}([e])$ and $\mathbf{V}([e'])$ have to be computed only if the integer vectors attached to e and e' coincide, which is rarely the case. With $\prec_C = \prec_1$, we have to compute $V_1([e])$ and $V_1([e'])$ if the first components of the integer vectors are equal; we have to compute $V_2([e])$ and $V_2([e'])$ if $V_1([e]) = V_1([e'])$ and the second components of the integer vectors are equal, and so on.

6.2 Computing *GState*([*e*])

Whenever the current branching process is extended with a new event e, the state $GState([e])$ has to be computed in order to determine if e is a cut-off event or not. For that, we first compute the cut corresponding to $[e]$; the labels of the conditions of this cut are $GState([e])$. Recall that the cut corresponding to $[e]$ is given by $(Min \cup [e]^\bullet) \setminus {}^\bullet[e]$, which provides a procedure to compute it. However, since it is too costly to store $[e]$ for each event e, the procedure involves computing the events preceding e.

The additional structure of synchronous products allows to easily compute the cut of $[e]$ from the cuts of the immediate predecessors of e, i.e., of the input events of e's input conditions. Let us start with a definition and a lemma:

Definition 9. *Let $p = (s, e)$ be an i-place of a branching process. The* depth *$d(p)$ of p is recursively defined as follows:*

- *If $e = \bot$, then $d(p) = 0$;*
- *If $e = (\mathbf{t}, X)$, then let p' be the unique i-place of X; define $d(p) = d(p') + 1$.*

Lemma 2. *Let $C_1, \ldots, C_k$ be configurations such that $C = C_1 \cup \ldots \cup C_k$ is also a configuration. Let $\mathbf{c}_i$ be the cut corresponding to C_i, and let $\mathbf{c}$ be the cut corresponding to C. For every $1 \leq j \leq n$, $\mathbf{c}(j)$ is the unique condition of the set $\{\mathbf{c}_1(j), \ldots, \mathbf{c}_k(j)\}$ having maximal depth.*

Proof. Since all the elements of $\{\mathbf{c}_1(j), \ldots, \mathbf{c}_k(j)\}$ are j-places, they are causally related or in conflict (Proposition 1). Since C is a configuration, they cannot be in conflict, and so they are all causally ordered. It follows that they all have different depths (notice that not all of $\mathbf{c}_1(j), \ldots, \mathbf{c}_k(j)$ have to be different, but of course all elements of $\{\mathbf{c}_1(j), \ldots, \mathbf{c}_k(j)\}$ are different by definition of set). So $\mathbf{c}(j)$ is well defined. We prove that $\mathbf{c}(j)$ belongs to the cut of C, i.e., that $\mathbf{c}(j) \in (Min \cup C^\bullet) \setminus {}^\bullet C$.

Assume without loss of generality that $\mathbf{c}(j) = \mathbf{c}_1(j)$. Then we have $\mathbf{c}(j) \in (Min \cup C_1^\bullet) \setminus {}^\bullet C_1$. So $\mathbf{c}(j) \in (Min \cup C_1^\bullet)$, and so $\mathbf{c}(j) \in (Min \cup C^\bullet)$. It remains to prove $\mathbf{c}(j) \notin {}^\bullet C$. Assume the contrary. Then there exists an index i such that $\mathbf{c}(j) \in {}^\bullet C_i$. It follows that the depth of $\mathbf{c}_i(j)$ must be greater than the depth of $\mathbf{c}(j)$, a contradiction.

We can now compute the cut of an event e as follows:

Proposition 4. *Let $e = (\mathbf{t}, X)$ be an event, and let $e_1, \ldots, e_k$ be its immediate predecessors. The cut of $[e]$ can be computed in two steps as follows:*

- *Compute the cut of $[e_1] \cup \ldots \cup [e_k]$ using Lemma 2; let $\mathbf{c}$ be this cut;*
- *For each output place p of e: If p is an i-place then replace the i-place of $\mathbf{c}$ by p.*

Proof. Observe that the output places of e belong to the cut of $[e]$. The rest follows easily from Lemma 2 and the definitions.

Let us apply this Proposition to compute the cut of [16] in Figure 3. The immediate predecessors of event 16 are events 10 and 12. Their corresponding cuts are $\langle n, g\rangle$ and $\langle f, p\rangle$. We have $d(n) = 4, d(g) = 2$ and $d(f) = 2, d(p) = 4$. So the cut of $[10] \cup [12]$ is $\langle n, p\rangle$. Now, the second step says to replace n by t and p by u. So the final result is $\langle t, u\rangle$. The fact that this is also the set of output places of event 16 is a coincidence.

In order to apply Proposition 4, our implementation has to compute the depth of each place of the current branching process. Fortunately, this leads to no time overhead. Recall that in order to decide if $[e_1] \prec_C [e_2]$ we attach to each event e the vector $\langle |V_1([e])|, \ldots, |V_n([e])|\rangle$. It follows immediately from the definitions that the depth of an i-place with input event e is equal to $|V_i([e])|$.

6.3 Computing $PE(BP)$

The computation of $PE(BP)$ is the most time consuming part of the algorithm. The computation is performed by considering each global transition $\mathbf{t} \in T$ in turn, and computing the possible extensions of BP of the form $(\mathbf{t}, X)$. So the problem consists of finding all $X \subseteq P$ such that (a) X is labelled by ${}^\bullet\mathbf{t}$, and (b) X is a *co*-set. Since the places of BP can be easily indexed according to the states they are labelled with, we search among all sets X satisfying (a) for those satisfying also (b). The implementation stores the *co*-relation of the places contained in the current branching process. Therefore, whenever the process is extended by a new event e, it is necessary to compute the places of the process that are in *co*-relation with the output places of e (notice that these output places themselves build a *co*-set).

A first procedure to compute this set of places applies the definition of the concurrency relation. Take the set of all places of the branching process, and perform the following steps:

(1) remove all places which are causally related with $e^\bullet$, by iteratively computing e's immediate predecessors, their immediate predecessors and so on; mark along the way all the places having more than one successor;
(2) remove the successors of the marked places (not already removed in (1)) ; these are the places in conflict with $e^\bullet$;
(3) give as output the remaining set of places.

To illustrate this procedure, assume that the current branching process is the prefix of Figure 3 containing events $1, 2, \ldots 11$, and that event 12 is the new event. Step (1) removes $\{k, g, c, a, e, b\}$, and marks a and e. Step (2) removes $\{d, h, i, l, m\}$. Step (3) yields $\{f, j, n, o\}$.

In the worst case, these steps require to visit all nodes of the current branching process, and since they have to be carried out whenever a new event e is added, the cost can be high. In the rest of the section we give a more efficient procedure.

Proposition 5. *Let $e = (\mathbf{t}, X)$ be a possible extension of a branching process (P, E). Let p be an output place of e, and let $p' \in P$ be an arbitrary place. $p \, co \, p'$ holds if and only if p' is an output place of e different from p, or $x \, co \, p'$ for every $x \in X$.*

Proof. If $p = p'$ then we are done, and so we consider only the case $p \neq p'$. Since e is a possible extension, $p < p'$ cannot hold, and so we have $p \; co \; p' \iff \neg(p' < p \vee p \# p')$. So it suffices to prove:

$$(p' < p \vee p \# p') \iff (p' \notin e^\bullet) \wedge (\exists x \in X. x \leq p' \vee p' \leq x \vee x \# p')$$

($\Longrightarrow$) We prove four statements:

- $p' < p \Rightarrow p' \notin e^\bullet$. Obvious, because no two output places of e are causally related.
- $p' < p \Rightarrow \exists x \in X. p' \leq x$. Since p has e as unique input event, the path from p' to p must necessarily contain e, and so it must also contain some input place of e, i.e., some element of X.

- $p \# p' \Rightarrow p' \notin e^\bullet$. Obvious, because no two output places of e are in conflict.
- $p \# p' \Rightarrow \exists x \in X . x \leq p' \vee x \# p'$. Since $p \# p'$ there exist two paths from a condition p'' to p and p' sharing only p''. If $p'' \in X$, then we have $p'' < p'$, and by taking $x = p''$ we are done. If $p'' \notin X$, then the path from p'' to p contains some element x of X, and so $x \# p'$.

($\Longleftarrow$) We consider three cases:

- $p' \notin e^\bullet \wedge \exists x \in X . x \leq p'$. Then there exist two paths from x to p and p' sharing only x. So $p \# p'$.
- $p' \notin e^\bullet \wedge \exists x \in X . p' \leq x$. Then, since $x < p$, we have $p' \leq p$.
- $p' \notin e^\bullet \wedge \exists x \in X . x \# p'$. Since $x < p$ and $x \# p'$, we have $p \# p'$.

If we assume that the *co*-relation is updated whenever a new event is added to the current branching process (P, E), then at the point of adding a new event $e = (\mathbf{t}, X)$ we can assume that we already know whether $x \; co \; p'$ holds or not for every $x \in X$ and every $p' \in P$. Updating the relation is now a simple matter. The following procedure takes care of it.

Procedure Update($(P, E), co, e = (\mathbf{t}, X)$)
begin
    $places := P$;
        for every $p \in P$ **do**
            for every $x \in X$ **do**
                if $\neg(x \; co \; p)$ **then** $places := places \setminus \{p\}$ **endif**
            endfor
        endfor;
    $co := co \cup (e^\bullet \times e^\bullet) \cup (e^\bullet \times places) \cup (places \times e^\bullet)$
end

The operations in the procedure can be efficiently implemented using a bitvector $co(p)$ for each place p.

There is also an obvious improvement concerning recomputations of $PE(BP)$. The algorithm computes $PE(BP)$, extends β by one event, say e, and recomputes $PE(BP)$. This is very inefficient, since numerous possible extensions may be recomputed again and again. In fact, the only new possible extensions after the addition of e are those having e as immediate predecessor. When the first event of the queue of possible extensions is added to the current branching process, only new extensions having this event as immediate predecessor are computed and inserted in the queue.

7 Experimental results

The abstract algorithm of section 4.1 was originally introduced in [4] for systems modelled by Petri nets. The same paper contained performance measures of an implementation, called `Imp1` in the sequel. Since synchronous products can be given a Petri net semantics, as sketched in Section 3, `Imp1` can also be applied to

	Synch.Prod.		Imp1			Imp12	Imp2		
System	*Comp.*	*Trans.*	*Events*	Cut-offs	Time	Time	*Events*	cut-offs	Time
DPH(7)	15	121	40672	21427	623.79	117.57	19306	9693	22.39
ELEVATOR(4)	7	939	16935	7337	96.03	24.32	16935	7337	25.42
KEY(3)	8	133	6940	2921	16.38	3.57	7187	3032	2.44
MMGT(3)	7	172	5841	2529	7.88	2.61	5841	2529	2.18
Q(1)	18	194	8402	1173	44.34	12.67	8030	1125	10.21
RING(24)	48	264	12745	1082	152.42	33.90	10722	1082	34.70
RW(12)	25	313	49177	45069	69.95	22.61	49177	45069	83.54
BUFFER(240)	240	241	28921	1	7098.06	1980.78	28921	1	34.81
CYCLIC(1000)	2000	5999	8996	1001	1372.24	1338.36	8996	1001	63.83
SENTST(2000)	2005	2030	2191	40	311.81	186.65	2030	40	8.33

Table 1. Experimental results

synchronous products. So it is possible to compare the performances of `Imp1` and the implementation of Section 6, called `Imp2` in the sequel. The main differences between `Imp1` and `Imp2` are

(a) `Imp1` uses the adequate order of [4], while `Imp2` uses $\prec_2$;
(b) `Imp1` computes the concurrency relation by the three-step procedure described at the beginning of Section 6.3, while `Imp2` uses the *Update* procedure;
(c) `Imp1` computes *Marking*$([e])$ (the equivalent of *GState*$([e])$ in [4]) by means of a backward search, while `Imp2` uses the procedure derived from Prop. 4.

The differences (a) and (c) are inherent to the change of model: Petri nets for `Imp1`, and synchronous products for `Imp2`. On the contrary, the difference (b) is accidental: When `Imp1` was programmed, we had not found the *Update* procedure. So it makes sense to consider a third implementation, `Imp12`, which coincides with `Imp1` on (a) and (c), and with `Imp2` on (b).

We have chosen a set of benchmarks compiled by Corbett in [3]; for a description of the systems the reader is referred to [3] and [7]. All benchmarks are scalable. Table 1 displays the results of the experiments for some representative cases. The experiments were carried out on a Sun Ultra 60 (295 MHz UltraSPARC-II) with 640 MB RAM using Solaris 2.7. The displayed data are the number of components and the number of global transitions of the product, the number of events of the complete prefix, the number of cut-off events, and the computation time (in seconds). The size of the unfoldings for `Imp1` and `Imp12` is always the same, since they both use the same adequate order. The benchmarks above the double horizontal line have a large ratio cut-offs/events, corresponding to wide but shallow prefixes, while those below have a small ratio, corresponding to narrow and deep prefixes. The results indicate that `Imp2` is indeed more efficient than `Imp1`. A closer look, and a comparison with `Imp12`, indicates that:

– For large cut-off ratios, the speed-up factor lies between 3 and 5, and it is due to the new procedure for the computation of the concurrency relation.

– For small cut-off ratios, the speed-up factor is of 1 to 2 orders of magnitude, and it is due to the new order, and to the new procedure for computing *GState*.

These provisory conclusions still need to be tested on more examples.

8 Conclusions

We have adapted the unfolding technique to Arnold's synchronous products of transition systems. The fact that a synchronous product consists of a fixed number of communicating sequential components has been used to simplify the unfolding procedure. We have obtained adequate orders simple to define and simple to prove correct.

We mentioned in the introduction that Langerak and Brinksma have applied the unfolding technique to a CSP-like process algebra [5]. The algebra has more modelling power than synchronous products; in particular, it is able to model nested parallelism, which synchronous products cannot. The price to pay is a more complicated adequate order than $\prec_1$ or $\prec_2$, although simpler than the order of [4] for Petri nets. Together with ours, Langerak and Brinksma's paper gives strong evidence that the unfolding technique can be applied to any model of concurrency allowing for a notion of independent actions.

We have presented an efficient implementation of the abstract algorithm for the construction of a complete finite prefix, which improves on the implementation of [4]. The speed-ups can reach two orders of magnitude in very favourable cases. A speed-up factor of at least 3 to 5 is achieved in nearly all cases.

Acknowledgements

Many thanks to Walter Vogler and Rom Langerak for helpful remarks and discussions.

References

1. Bibliography on net unfoldings. Accessible at `http://wwwbrauer.informatik.tu-muenchen.de/~esparza`.
2. A. Arnold. *Finite Transition Systems*. Prentice-Hall, 1992.
3. J.C. Corbett. Evaluating Deadlock Detection Methods for Concurrent Software. In T. Ostrand, editor, *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, 204–215, 1994.
4. J. Esparza, S. Römer und W. Vogler: An Improvement of McMillan's Unfolding Algorithm. In T. Margaria und B. Steffen, editors, *Proceedings of TACAS'96*, LNCS 1055, 87–106, 1996.
5. R. Langerak and E. Brinksma. A Complete Finite Prefix for Process Algebra. To appear in Proceedings of CAV '99, 1999.
6. K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer, 1993.
7. S. Melzer. Verification of Distributed Systems Using Linear and Constraint Programming. Ph. D. Thesis, Technical University of Munich, 1998 (in German).