A Complete Finite Prefix for Process Algebra

Rom Langerak and Ed Brinksma

University of Twente, Department of Computer Science, PO Box 217, 7500 AE Enschede, The Netherlands, {langerak,brinksma}@cs.utwente.nl

Abstract. In this paper we show how to use McMillan's complete finite prefix approach for process algebra. We present the model of component event structures as a semantics for process algebra, and show how to construct a complete finite prefix for this model. We present a simple adequate order (using an order on process algebra expressions) as an optimization to McMillan's original algorithm.

1 Introduction

A major problem in the verification of distributed systems is the state explosion problem. This problem results when the modelling a system consisting of parallel subsystems causes the model to have a number of states that is of the same order of magnitude as the product of the states of the subsystems.
In process algebra (e.g. [Hoa85,BB87,BW90] state explosion may occur when using the standard interleaving semantics. In order to deal with this problem one line of research has been to look for alternative semantic models based on partial orders, of which event structures [Win89,BC94,Lan92] are a prominent example. Event structures can be used as a semantics for process algebra and are easily extended with timing, probabilistic and stochastic information [BKLL98, KLL$^+$98]. A problem though with event structures is that recursion leads to infinite structures, whereas for techniques like model checking it is important to have finite representations of infinite behaviour.
An interesting direction of research has been initiated by McMillan, originally for finite state Petri nets [McM92,McM95a,McM95b]. He has presented an algorithm for constructing an initial part of the occurrence net [NPW81,Eng91] of a Petri net which contains all information on reachable states and transitions. This so-called complete finite prefix can be used as the basis for model checking [Esp94, Gra97,Wal98].
In this paper we explore how this McMillan complete finite prefix approach can be used in giving an event structure semantics to process algebra. Using a translation of process algebra into Petri nets (as has been done in [Old91]) would pose severe complications when calculating a prefix. The translation there makes use of a trick for dealing with the choice operator; this has as a side effect that not all reachable markings correspond in a clear way to reachable process algebra expressions. This would greatly complicate the computation of a finite prefix. Therefore we directly translate a process algebra expression into a model similar to an occurrence net in which choice can be modelled in a natural way.

N. Halbwachs and D. Peled (Eds.): CAV'99, LNCS 1633, pp. 184–195, 1999.

The paper is organized as follows. In section 2 we present a process algebra and a model called *component event structures* which is to process algebra what occurrence nets are to Petri nets. In section 3 we use this model as a semantics for process algebra and in section 4 we show how to construct a complete finite prefix for this model. In section 5 we present an optimization to the McMillan algorithm which has the same advantages as a proposal in [ERV97] but profits from the process algebra setting. Section 6 is for conclusions.
An extended version of this paper (containing all proofs) can be found in [LB99].

2 Process Algebra and Component Event Structures

This paper uses a simple process algebra with a parallel operator similar to the one from CSP [Hoa85] or LOTOS [BB87]. The syntax is given by the following grammar:

$B ::= \mathbf{stop} \mid a; B \mid B + B \mid B \,|_A\, B \mid P$

The *inaction* process **stop** cannot do anything. *Action prefix* is denoted by $a; B$ where $a \in \mathit{Act}$, with Act a set of *actions* (a distinction between observable and invisible actions plays no role in this paper). The *choice* between B_1 and B_2 is denoted by $B_1 + B_2$. *Parallel composition* is denoted by $B_1 \,|_A\, B_2$ where A is the set of synchronizing actions; $B_1 \,|_\emptyset\, B_2$ is abbreviated to $B_1 | B_2$. Finally, P denotes *process instantiation* where a behaviour expression is assumed to be in the context of a set of process definitions of the form $P := B$ with B possibly containing process instantiations of P.

A process algebra expression can be decomposed into so-called components, which are action prefix expressions together with information about the synchronization context. This approach has been inspired by the Petri net semantics for process algebra presented in [Old91].

Definition 1. A *component* s is defined by
$S ::= \mathbf{stop} \mid a; B \mid S\,|_A \mid |_A\, S$ with B a process algebra expression; the universe of all components is denoted by Comp.

Convention: let $\mathcal{S} = \{S_1,\ \ldots,\ S_n\}$ be a set of components, then we use the notation $\mathcal{S}\,|_A = \{S_1\,|_A,\ \ldots,\ S_n\,|_A\}$, and similarly for $|_A\,\mathcal{S}$.

Components can be in a *choice* relation which will be used to model the effect of the process algebra choice operator.

Definition 2. A *state* is a tuple $(\mathcal{S},\ \mathcal{R})$ with $\mathcal{S}$ a set of components, and $\mathcal{R}$ an irreflexive and symmetric relation between components (so $\mathcal{R} \subset \mathcal{S} \times \mathcal{S}$) called the *choice* relation.

Convention: let $\mathcal{R} = \{(S_1,\ S_1'),\ \ldots,\ (S_n,\ S_n')\}$ be a choice relation, then we use the notation $\mathcal{R}\,|_A = \{(S_1\,|_A,\ S_1'\,|_A),\ \ldots,\ (S_n\,|_A,\ S_n'\,|_A)\}$ and similarly for $|_A\,\mathcal{R}$.
Components (and the choice relation between them) can be obtained by *decomposing* a process algebra expression.

Definition 3. The decomposition function *dec*, which maps a process algebra expression on a state, is recursively defined by $dec(B) = (\mathcal{S}(B),\ \mathcal{R}(B))$ with

$dec(\mathbf{stop}) = (\{\mathbf{stop}\},\ \emptyset)$
$dec(a_x; B) = (\{a_x; B\},\ \emptyset)$
$dec(B \mid_A B') = (\mathcal{S}(B)|_A \cup |_A\mathcal{S}(B'),\ \mathcal{R}(B)|_A \cup |_A\mathcal{R}(B'))$
$dec(B + B') = (\mathcal{S}(B) \cup \mathcal{S}(B'),\ \mathcal{R}(B) \cup \mathcal{R}(B') \cup (\mathcal{S}(B) \times \mathcal{S}(B')))$
$dec(P_\Phi) = dec(\Phi(B))$ if $P := B$

In order to avoid that the decomposition of a process instantiation leads to an infinite chain of substitutions, we have to adopt the constraint that all process definitions are guarded (see e.g. [BW90]).

We define an event structure model which is very similar to a type of Petri nets called *occurrence* nets [NPW81,Eng91]; the main difference is that there are no tokens, and conditions can be in a binary *choice* relation.

Definition 4. A *condition event structure* is a 4-tuple $(D,\ E,\ \sharp,\ \prec)$ with:

- D a set of conditions
- E a set of events
- $\sharp \subset D \times D$, the choice relation (symmetric and irreflexive)
- $\prec\ \subseteq (D \times E) \cup (E \times D)$ the *flow* relation

We adopt some Petri net terminology: a *marking* is a set of conditions. A *node* is either a condition or an event. The *preset* of a node x, denoted by ${}^\bullet x$, is defined by ${}^\bullet x = \{y \in D \cup E \mid y \prec x\}$, and the *postset* $x^\bullet$ by $x^\bullet = \{y \in D \cup E \mid x \prec y\}$. The *initial marking* M_0 is defined by $M_0 = \{d \in D \mid {}^\bullet d = \emptyset\}$.

Definition 5. The transitive and reflexive closure of $\prec$ is denoted by $\leq$.
The *conflict* relation on nodes, denoted by $\#$, is defined by: let x_1 and x_2 be two different nodes, then $x_1 \# x_2$ iff there are two nodes y_1 and y_2, such that $y_1 \leq x_1$ and $y_2 \leq x_2$, with

- either y_1 and y_2 are two conditions in the choice relation, i.e. $y_1 \sharp y_2$
- or y_1 and y_2 are two events with ${}^\bullet y_1\ \cap\ {}^\bullet y_2 \neq \emptyset$

Definition 6. A condition event structure is *well-formed* if the following properties hold:

1. $\leq$ is anti-symmetric, i.e. $x \leq x' \wedge x' \leq x \Rightarrow x = x'$
2. finite precedence, i.e. for each node x the set $\{y \in E \cup D \mid y \leq x\}$ is finite
3. no self-conflict, i.e. for each node x: $\neg(x \# x)$
4. for each event e: ${}^\bullet e \neq \emptyset$ and $e^\bullet \neq \emptyset$
5. for each condition d: $|{}^\bullet d| \leq 1$
6. for all conditions d_1 and d_2: $d_1 \sharp d_2 \Rightarrow\ {}^\bullet d_1 =\ {}^\bullet d_2$

Let d be a condition, then we define $\sharp(d)$, the set of conditions in choice with d, by $\sharp(d) = \{d' \mid d \sharp d'\}$. Similarly for a set of conditions $\mathcal{D}$, $\sharp(\mathcal{D}) = \{d' \mid \exists d \in \mathcal{D} : d \sharp d'\}$.

Definition 7. Suppose we have a condition event structure, with e an event, and M and M' markings, then we say there is an *event transition* $M \xrightarrow{e} M'$ iff ${}^\bullet e \subseteq M$ and $M' = (M \cup e^\bullet) \setminus ({}^\bullet e \cup \sharp({}^\bullet e))$ (note there are no loops in well-formed condition event structures).
An *event sequence* is a sequence of events $e_1 \dots e_n$ such that there are markings $M_1, \ \dots, \ M_n$ with $M_0 \xrightarrow{e_1} M_1 \longrightarrow \dots \xrightarrow{e_n} M_n$. We call $C = \{e_1, \ \dots, \ e_n\}$ a configuration of the condition event structure.

Definition 8. Two nodes x and x' are said to be independent, notation $x \asymp x'$, iff $\neg(x \leq x') \wedge \neg(x' \leq x) \wedge \neg(x \# x')$.

Definition 9. A *cut* is a marking M such that for each pair of different conditions d and d' in M holds: $d \asymp d'$ or $d \sharp d'$, and that is maximal (w.r.t. set inclusion).

Theorem 1. *Let C be a configuration and M a cut. Define $Cut(C) = (M_0 \cup C^\bullet) \setminus ({}^\bullet C \cup \sharp({}^\bullet C))$ and $Conf(M) = \{e \in E \mid \exists d \in M : e \leq d\}$. Then: $Cut(C)$ is a cut, $Conf(M)$ is a configuration, $Conf(Cut(C)) = C$, and $Cut(Conf(M)) = M$.*

Definition 10. A condition event structure $\mathcal{E} = (D, \ E, \ \sharp, \ \prec)$ with mappings
$l_C : D \rightarrow Comp$ (mapping conditions to components)
$l_E : E \rightarrow Act$ (mapping events to actions)
is called a *component event structure.*

We will often be sloppy and denote a condition by its component label (but note that different conditions may be labelled with the same component).

3 Component Event Structures as Semantics for Process Algebra

In this section we define a component event structure as a semantics for a process algebra expression with the help of a derivation system for transitions (again inspired by [Old91]). This derivation system will allow derivations of transitions of the form $\mathcal{S} \xrightarrow{a} (\mathcal{S}', \ \mathcal{R}')$, where $\mathcal{S}$ and $\mathcal{S}'$ are sets of components, and $\mathcal{R}'$ is a choice relation over components $\mathcal{S}'$. The rules are given in table 1.

Definition 11. Let $\mathcal{E}$ be a component event structure. The *possible extensions* of $\mathcal{E}$, denoted by $PE(\mathcal{E})$, is the set of all pairs $(\mathcal{D}, \ \mathcal{S} \xrightarrow{a} (\mathcal{S}', \ \mathcal{R}'))$ such that:
- $\mathcal{D}$ is a set of pairwise independent conditions of $\mathcal{E}$, with $l_D(\mathcal{D}) = \mathcal{S}$
- $\mathcal{S} \xrightarrow{a} (\mathcal{S}', \ \mathcal{R}')$ can be derived from the rules in table 1
- $\mathcal{E}$ does not already contain an event e with $l_E(e) = a$ and ${}^\bullet e = \mathcal{D}$

For component event structures it is easy to check that if two conditions have the same component label they are in conflict; this means that a set of pairwise independent conditions is labelled by a set of components with the same cardinality.

Table 1. Derivation system for component transitions

$$\{a_x; B\} \xrightarrow{a} dec(B)$$

$$\frac{\mathcal{S} \xrightarrow{a} (\mathcal{S}',\ \mathcal{R}')}{\mathcal{S}|_A \xrightarrow{a} (\mathcal{S}'|_A,\ \mathcal{R}'|_A)}\ (a \notin A) \qquad \frac{\mathcal{S} \xrightarrow{a} (\mathcal{S}',\ \mathcal{R}')}{|_A\mathcal{S} \xrightarrow{a} (|_A\mathcal{S}',\ |_A\mathcal{R}')}\ (a \notin A)$$

$$\frac{\mathcal{S}_1 \xrightarrow{a} (\mathcal{S}_1',\ \mathcal{R}_1') \quad \mathcal{S}_2 \xrightarrow{a} (\mathcal{S}_2',\ \mathcal{R}_2')}{\mathcal{S}_1|_A \cup |_A\mathcal{S}_2 \xrightarrow{a} (\mathcal{S}_1'|_A \cup |_A\mathcal{S}_2',\ \mathcal{R}_1'|_A \cup |_A\mathcal{R}_2')}\ (a \in A)$$

We can *add* a possible extension $(\mathcal{D},\ \mathcal{S} \xrightarrow{a} (\mathcal{S}',\ \mathcal{R}')) \in PE(\mathcal{E})$ to $\mathcal{E}$ by adding a new event e labelled a and new conditions $\mathcal{D}'$ with labels from $\mathcal{S}'$, such that ${}^\bullet e = \mathcal{D}$ and $e^\bullet = \mathcal{D}'$, and a choice relation over the conditions $\mathcal{D}'$ induced by the relation $\mathcal{R}'$ over $\mathcal{S}'$.

Algorithm 1. Let B be a process algebra expression, with $dec(B) = (\mathcal{S}_0, \mathcal{R}_0)$. The *unfolding* of B, denoted $Unf(B)$, is generated by the following algorithm:

```
Let $\mathcal{E}$ be the component event structure with conditions $M_0$,
$l_D(M_0) = \mathcal{S}_0$, choice relation $\mathcal{R}_0$, and no events;
$pe := PE(\mathcal{E})$;

while $pe \neq \emptyset$
do select a pair $(\mathcal{D},\ \mathcal{S} \xrightarrow{a} (\mathcal{S}',\ \mathcal{R}'))$ from $pe$;
    add it to $\mathcal{E}$;
    $pe := PE(\mathcal{E})$
od;
$Unf(B) = \mathcal{E}$
```
□

The algorithm only terminates for expressions with finite behaviour. For expressions with infinite behaviour, the above algorithm produces arbitrarily large unfolding approximations (under the fairness assumption that each pair in *pe* is eventually added to $\mathcal{E}$). In that case we define $Unf(B)$ as the limit of these approximations. It is easy to prove that $Unf(B)$ is a well-formed component event structure, i.e. the properties of definition 6 hold.
Notation: let $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$, and $\mathcal{S}' \subseteq \mathcal{S}$, then $\mathcal{R}\lceil\mathcal{S}' = \mathcal{R} \cap (\mathcal{S}' \times \mathcal{S}')$. Note that if $\sharp$ is the choice relation of $Unf(B)$, and $dec(B) = (M,\ \mathcal{R})$, then by the definition of unfolding $\mathcal{R} = \sharp\lceil M$.
In [Lan92] it is shown how by slightly adapting the standard operational semantics it is possible to derive event sequences. In [LB99] this idea has been adapted to component event structures and the following result has been proven there.

Theorem 2. *Let B be a process algebra expression, $Unf(B)$ its unfolding and M_0 the initial marking of $Unf(B)$. Let σ be an event trace. Then:*

$$B \xrightarrow{\sigma} B' \Leftrightarrow M_0 \xrightarrow{\sigma} M' \qquad \textit{with } dec(B') = (M', \sharp\lceil M')$$

In the last section we saw that there is a one-to-one correspondence between cuts and configurations via the mappings *Cut* and *Conf*. In [Lan92] it has been proven (Theorem 7.4.1) that there is also a one-to-one correspondence between configurations and reachable states (where each reachable state is a process algebra expression). It follows that there is a one-to-one correspondence between cuts and states of some unfolding $Unf(B)$; therefore given a cut M', there is a process algebra expression B' such that $dec(B') = (M', \sharp\lceil M')$. So given an unfolding $Unf(B)$, we define a mapping *St* from cuts to process algebra expressions by $St(M') = B'$ where B' is the reachable state corresponding to $Conf(M')$, so $dec(B') = (M', \ \sharp\lceil M')$. If C is a configuration, we will also write $St(C)$ for $St(Cut(C))$.

4 A Complete Finite Prefix for Component Event Structures

In the last section we have defined the component event structure $Unf(B)$ for a process algebra expression B. This representation may be infinite for recursive processes; we would like to have a finite representation of such behaviour.
Therefore in this section we will look at McMillan's so-called *complete finite prefix* of an unfolding, which is an initial part of the unfolding that is complete in the following sense:
For each cut M of $Unf(B)$ there is a cut M' of the finite prefix such that:

- $St(M) = St(M')$, so the prefix contains all reachable states
- if $M \xrightarrow{e}$ in $Unf(B)$ with $l_E(e) = a$ then $M' \xrightarrow{e'}$ and $l_E(e') = a$, so the prefix contains all transitions

The complete finite prefix and McMillan's algorithm for computing it have originally been defined in the context of Petri nets (see [McM95b,Esp94,ERV97]). However, the approach (using the concepts of event, configuration and cut) can be transferred completely to the setting of component event structures, as we show here. For details and proofs we refer to [McM95b,Esp94,ERV97].

The complete finite prefix approach only works for finite state processes, i.e. processes with a finite number of reachable states. It is in general undecidable whether a process algebra expression is finite state. However, there exist syntactical restrictions that are sufficient to guarantee that an expression is finite state (see [FGM92] for discussion and overview). In the following we simply assume that all process algebra expressions are finite state.

We first need some preliminary definitions where we closely follow [ERV97].
Let E be a set of events and let C be a configuration of a component event structure. If $C \cup E$ is a configuration, and $C \cap E = \emptyset$, then we denote $C \cup E$ by $C \oplus E$, the *extension* of C by E.
Let M be a marking of a (well-formed) component event structure. Define the successor nodes of M by $N = \{x \in E \cup D \mid \exists y \in M : y \leq x\}$. We define $\Uparrow M = (D \cap N, \ E \cap N, \ \sharp\lceil N, \ \prec \lceil N)$. It is easy to check that $\Uparrow M$ is a well-formed component event structure.

It is easy to check that for a configuration C the unfolding $\mathit{Unf}(\mathit{St}(C))$ is isomorphic to $\Uparrow \mathit{Cut}(C)$. So if C_1 and C_2 are two configurations such that $\mathit{St}(C_1) = \mathit{St}(C_2)$, then $\Uparrow \mathit{Cut}(C_1)$ and $\Uparrow \mathit{Cut}(C_2)$ are isomorphic. So there is an isomorphism $I_{C_1}^{C_2}$ from $\Uparrow \mathit{Cut}(C_1)$ to $\Uparrow \mathit{Cut}(C_2)$; this induces a mapping from the extensions of C_1 onto the extensions of C_2, so $C_1 \oplus E$ is mapped onto $C_2 \oplus I_{C_1}^{C_2}(E)$.

The following definition presents an important technical aspect of the calculation of a complete finite prefix.

Definition 12. A (strict) partial order $\sqsubset$ on the finite configurations of an unfolding is an *adequate order* iff:
1. $\sqsubset$ is well-founded, i.e. there is no infinite sequence $C_1 \sqsupset C_2 \sqsupset \ldots$
2. $\sqsubset$ refines $\subset$, i.e. $C_1 \subset C_2$ implies $C_1 \sqsubset C_2$
3. $\sqsubset$ is preserved by finite extensions, which means that if $C_1 \sqsubset C_2$ and $\mathit{St}(C_1) = \mathit{St}(C_2)$, then $C_1 \oplus E \sqsubset C_2 \oplus I_{C_1}^{C_2}(E)$.

The original algorithm by McMillan uses as adequate order the order $\sqsubset_m$ defined by $C_1 \sqsubset_m C_2 \Leftrightarrow |C_1| < |C_2|$. This order is intuitively easy to understand but can be very inefficient. An improvement has been given in [ERV97]; in the next section we present an adequate order that is very suitable for a process algebra prefix.

Let e be an event of a component event structure, then the *local configuration* $[e]$ is defined by $[e] = \{e' \in E | e' \leq e\}$ (it is very easy to prove that $[e]$ is indeed a configuration).

Definition 13. Let $\mathit{Unf}(B)$ be an unfolding and let $\sqsubset$ be the selected adequate partial order on the configurations of $\mathit{Unf}(B)$. An event e is a *cut-off event* if $\mathit{Unf}(B)$ has a local configuration $[e']$ such that $\mathit{St}([e]) = \mathit{St}([e'])$ and $[e'] \sqsubset [e]$

Definition 14. Let X be the set of nodes of $\mathit{Unf}(B)$ such that $x \in X$ iff no event causally preceding x is a cut-off event. Then the *finite prefix* $\mathit{Fp}(B)$ of $\mathit{Unf}(B) = (D, E, \sharp, \prec)$ is defined by $\mathit{Fp}(B) = (D \cap X,\ E \cap X,\ \sharp\lceil X,\ \prec \lceil X)$

So $\mathit{Fp}(B)$ contains all local configurations, and stops at cut-off events since their local configuration has been encountered already. The nice result (originally proven by McMillan for Petri nets [McM95b]) is that this is enough to guarantee completeness, so the prefix contains also all non-local configurations; $\mathit{Fp}(B)$ is finite and complete.

Conceptually a finite prefix is obtained by taking an unfolding and cutting away all successor nodes of cut-off events. This is not a practical recipe; the next algorithm shows how to obtain directly the complete finite prefix, without first creating the (possibly infinite) unfolding. First we redefine the set of possible extensions, to make sure that no successors of cut-off events are created.

Definition 15. Let $\mathcal{E}$ be a labelled component event structure with a set of cut-off events *cut*. The *possible non-cut-off extensions* of $\mathcal{E}$, denoted by $PE'(\mathcal{E}, \mathit{cut})$, is the set of all pairs $(\mathcal{D},\ \mathcal{S} \xrightarrow{a} (\mathcal{S}',\ \mathcal{R}'))$ such that $(\mathcal{D},\ \mathcal{S} \xrightarrow{a} (\mathcal{S}',\ \mathcal{R}')) \in PE(\mathcal{E})$ and $\forall d \in \mathcal{D} :\ {}^\bullet d \notin \mathit{cut}$

Algorithm 2. Let B be a process algebra expression, with $dec(B) = (\mathcal{S}_0, \mathcal{R}_0)$. Then the finite prefix $Fp(B)$, is generated by the following algorithm:

```
Let $\mathcal{E}$ be the component event structure with components $M_0$,
$l_D(M_0) = \mathcal{S}_0$, choice relation $\mathcal{R}_0$, and no events;
$cut := \emptyset$;
$pe := PE'(\mathcal{E}, cut)$;

**while** $pe \neq \emptyset$
**do** select a pair $(\mathcal{D},\ \mathcal{S} \xrightarrow{a} (\mathcal{S}',\ \mathcal{R}'))$ from $pe$ such that adding it
          leads to a new $e$ with $[e]$ minimal w.r.t. $\sqsubset$;
    add it to $\mathcal{E}$;
    if $e$ is a cut-of event then $cut := cut \cup \{e\}$;
    $pe := PE'(\mathcal{E}, cut)$
**od**;
$Fp(B) = \mathcal{E}$
```
□

It is easy to check that $Fp(B)$ as generated by algorithm 2 contains all nodes of $Unf(B)$ that are not causally preceded by a cut-off event, so it is indeed the finite prefix defined by definition 14.

Example 1. Consider $B = P \mid_b Q$ with $P = a; b; P$ and $Q = c; b; (e; P + d; Q)$. Then the unfolding is given in figure 1; cut-off events are indicated by a box.

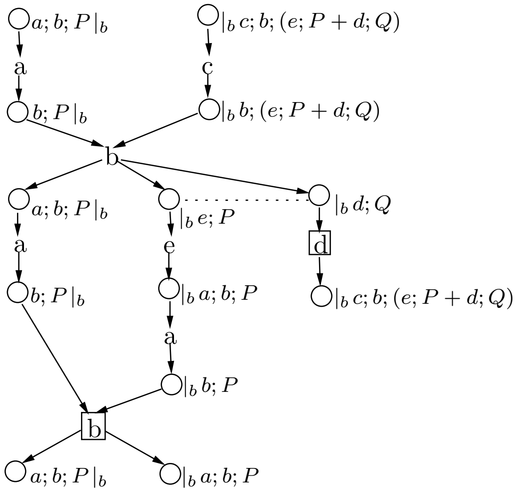

Fig. 1. Example of a complete finite prefix

5 An Adequate Order for Process Algebra

As already noted in [ERV97], the original McMillan ordering $\sqsubset_m$ defined by $C_1 \sqsubset_m C_2$ iff $|C_1| < |C_2|$ can be quite inefficient. Consider e.g. the expression $a; P + b; P$. Now although both a and b lead to the same state, it is not possible to make one of them a cut-off event as $[a]$ and $[b]$ have the same number of events. This makes it possible to find examples in which the finite prefix has a size that is exponential in the number of reachable states of the process algebra expression.
In [ERV97] an adequate order has been defined that does not suffer from this problem. This order is total on all configurations, so whenever two local configurations have the same state this leads to a cut-off. This is an important improvement on the original McMillan order, but an adequate order does not need to be total on all configurations, in order to have this property. The order in [ERV97] is rather complicated as it requires operations on configurations like subtracting the set of minimal events of a configuration (in fact this order is defined on suffixes of configurations).

In this section we define an adequate order which differs from $\sqsubset_m$ only for configurations having the same state and the same number of events. This order is easy to implement as it is defined syntactically as a kind of lexicographical order on process algebra expressions. The order orders each pair of configurations with the same state, so local configurations having the same state always lead to a cut-off.

We assume that initially process instantiations in an expressions are indexed with *simple process indices* (denoted by Greek letters) and actions are indexed by *action indices.* We furthermore assume an operation $\Phi(B)$ that takes an indexed expression B and prefixes all indices with Φ. We change the operational rule for process instantiation into $(\Phi(B) \xrightarrow{a} B', P := B) \Rightarrow P_\Phi \xrightarrow{a} B')$. This means that the process index of an instantiation P_Φ has the effect of prefixing all indices in the defining expression of P with Φ, leading to process indices that are strings of simple process indices; for details we refer to [LB99].

We assume there is an order on simple process indices; this order is arbitrary but we have (for technical reasons) the constraint that the order should respect the left to right order of the indices in the indexed process algebra expression that we are interested in (so if α is ordered before β, it will occur as a process index to the left of the occurrence of β). Remember that a proces index is a string of simple process indices; so the order on simple process indices induces a lexicographical order on process indices. This lexicographical order is not well-founded but can be used to define a well-founded order on process indices:

Definition 16. Let Φ_1 and Φ_2 be two process indices. We define $\Phi_1 \ll \Phi_2$ iff either $|\Phi_1| < |\Phi_2|$, or $|\Phi_1| = |\Phi_2|$ and Φ_1 is lexicographically smaller than Φ_2.

With this order we can define an order on process algebra expressions that are equal modulo process identifiers; B_1 and B_2 are equal modulo process indentifier, notation $B_1 =_p B_2$, iff after removing all process indices they are equal.

A component is of the form $\mathbf{stop}_{\Phi}$ or $a_{\Phi i}; B$, with Φ a process index and i a simple action index, possibly decorated with strings of parallel operators to the left and right. In these cases we call Φ the process index of the component.

Definition 17. Let B_1 and B_2 be two different process algebra expressions with $B_1 =_p B_2$. Then to each component of B_1 corresponds a component of B_2 that is equal modulo process identifiers. We define $B_1 \ll B_2$ iff for the leftmost first two corresponding components of B_1 respectively B_2 that have different process indices Φ_1 respectively Φ_2 holds: $\Phi_1 \ll \Phi_2$.

Example 2. Suppose α comes before β in the order on simple process indices, then $a_{\alpha 1}; P_{\alpha\phi} \,|_G\, b_{\alpha 2}; Q_{\alpha\psi} \ll a_{\alpha 1}; P_{\alpha\phi} \,|_G\, b_{\beta 2}; Q_{\beta\psi}$ and
$\mathbf{stop}_{\alpha} \,|_G\, a_1; P_{\phi} \ll \mathbf{stop}_{\beta} \,|_G\, a_1; P_{\phi}$

With the help of $\ll$ we define the *state* order $\sqsubset_s$ on the configurations of $\mathit{Unf}(B)$:

Definition 18. Let C_1 and C_2 be two configurations of $\mathit{Unf}(B)$. Then $C_1 \sqsubset_s C_2$ iff $|C_1| < |C_2|$ or: $|C_1| = |C_2|$, $\mathit{St}(C_1) =_p \mathit{St}(C_2)$ and $\mathit{St}(C_1) \ll \mathit{St}(C_2)$.

Theorem 3. *$\sqsubset_s$ is an adequate order on the configurations of $\mathit{Unf}(B)$.*

Just like the adequate order presented in [ERV97] (denoted $\prec_r$ there) our order has the property that for each pair of events e and e' with $\mathit{St}([e]) = \mathit{St}([e'])$: either $[e] \sqsubset_s [e']$ or $[e'] \sqsubset_s [e]$. This has two desirable consequences:

– the number of non-cut-off events in a complete finite prefix cannot exceed the number of local states (i.e. states of local configurations)
– since events are generated in accordance with $\sqsubset_s$ in algorithm 2, we need for each newly added event e only to check if there is already an event e' with $\mathit{St}([e]) = \mathit{St}([e'])$ in order to check that e is a cut-off event.

We think that in comparison with the adequate order of [ERV97] our order is easier to understand as it is based on a syntactical lexicographical order on process algebra expressions. For the same reason we expect it to be easy to implement. This will be checked in an implementation of our algorithm that is currently under construction.

6 Conclusions

We have presented component event structures which are similar to both prime event structures and occurrence nets. The advantage of component event structures over Petri nets is that the choice operator can be modelled naturally with the choice relation. When using Petri nets to model process algebra (as has been done in [Old91]) extra places have to be introduced, using a technical trick, to model the effect of choice. This trick leads to markings that do not directly correspond to process algebra expressions (only after a kind of garbage collection) which would greatly complicate the construction of a complete finite prefix. Our component event structures do not suffer from these complications. In addition, they are very similar to prime event structures which can be obtained by just deleting the components.

We have shown how McMillans approach can be used for obtaining a finite complete prefix. We have presented an optimization that has the same effect as the one in [ERV97] but profits from the process algebra context in such a way that it is less complex.

Our current research is concentrating on how the complete prefix can be transformed into a kind of graph grammar that produces the infinite behavior. This graph grammar representation can then be used for simulation and model checking. Furthermore, using timed, probabilistic and stochastic extensions similar to [KLL$^+$98,BKLL98] we will investigate how the graph grammar can be used for performance modelling. We are also working on an implementation which we hope to finish soon.

Acknowledgements

Many thanks to Joost-Pieter Katoen, Diego Latella and Mieke Massink for extensive discussions and many suggestions for improvement.

References

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BC94] G. Boudol and I. Castellani. Flow models of distributed computations: three equivalent semantics for CCS. *Information and Computation*, 114:247–314, 1994.

[BKLL98] E. Brinksma, J.-P. Katoen, D. Latella, and R. Langerak. Partial-order models for quantitative extensions of LOTOS. *Computer Networks and ISDN Systems*, 30(9/10):925–950, 1998.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[Eng91] J. Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28:575–591, 1991.

[ERV97] J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In *Proc. TACAS '96*, volume 1055 of *Lecture Notes in Computer Science*, pages 87–106. Springer-Verlag, 1997.

[Esp94] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2):151–195, 1994. Also appeared in *Proc. TAPSOFT '93*, volume 668 of *Lecture Notes in Computer Science*, pages 613–628. Springer-Verlag, 1993.

[FGM92] A. Fantechi, S. Gnesi, and G. Mazzarini. The expressive power of LOTOS behaviour expressions. Nota Interna I.E.I. B4-43, I.E.I (Pisa), October 1992.

[Gra97] B. Graves. Computing reachability properties hidden in finite net unfoldings. *Lecture Notes in Computer Science*, 1055:327–342, 1997.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[KLL$^+$98] J.-P. Katoen, D. Latella, R. Langerak, E. Brinksma, and T. Bolognesi. A consistent causality-based view on a timed process algebra including urgent interactions. *Journal on Formal Methods for System Design*, 12(2):189–216, 1998.

[Lan92] R. Langerak. *Transformations and Semantics for LOTOS.* PhD thesis, University of Twente, 1992.

[LB99] R. Langerak and E. Brinksma. A complete finite prefix for process algebra. Technical report, University of Twente, January 1999.

[McM92] K. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. CAV '92, Fourth Workshop on Computer-Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 164–174, 1992.

[McM95a] K. McMillan. Trace theoretic verification of asynchronous circuits using unfoldings. In *Proc. CAV '95, 7th International Conference on Computer-Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 180–195. Springer-Verlag, 1995.

[McM95b] K.L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6:45 – 65, 1995.

[NPW81] M. Nielsen, G.D. Plotkin, and G. Winskel. Petri nets, event structures and domains, part 1. *Theoretical Computer Science*, 13(1):85–108, 1981.

[Old91] E.-R. Olderog. *Nets, terms and formulas.* Cambridge University Press, 1991.

[Wal98] F. Wallner. Model-checking LTL using net unfoldings. In *Proc. CAV '98, 10th International Conference on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 207–218, Vancouver, Canada, 1998.

[Win89] G. Winskel. An introduction to event structures. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, Lecture Notes in Computer Science, pages 364–397. Springer-Verlag, 1989.