Deriving a Graph Rewriting System from a Complete Finite Prefix of an Unfolding

Rom Langerak [1]

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

Abstract

The starting point of this paper is McMillan's complete finite prefix of an unfolding that has been obtained from a Petri net or a process algebra expression. The paper addresses the question of how to obtain the (possibly infinite) system behaviour from the complete finite prefix. An algorithm is presented to derive from the prefix a graph rewriting system that can be used to construct the unfolding. It is shown how to generate event sequences from the graph rewriting system which is important for constructing an interactive simulator. Finally it is indicated how the graph rewriting system yields a transition system that can be used for model checking and test derivation.

1 Introduction

In order to deal with the state explosion problem in validating distributed systems, many alternatives to the standard interleaving semantics have been proposed. A large class of them can be classified as partial order semantics, of which several types of event structures [Win89,BC94,Lan92] and occurrence nets [NPW81,Eng91] are prominent examples. A problem with these models is that in general recursion is dealt with via a fixed point technique leading to infinite structures, whereas one would like to have finite representations, especially for computer aided verification.

An interesting direction of research has been initiated by McMillan, originally for finite state Petri nets [McM92,McM95a,McM95b]. He has presented an

[1] This work has been partially funded by Progetto Trilaterale CNR - Quantitative Event Structures, Discrete Simulation and Performance Modeling, and the TI Project: Systems Validation Centre.

algorithm that for a given Petri net constructs an initial part of the special occurrence net called unfolding or maximal branching process [NPW81,Eng91]. This so-called *complete finite prefix* contains all information on reachable states and transitions. An important optimization has been defined in [ERV97]. In [LB99] the complete finite prefix approach has been adapted for process algebra (together with an efficient optimization) for a model similar to occurrence nets called *condition event structures*.

If the complete finite prefix contains all information on the (possibly infinite) system behaviour, how to make use of this information? How can the system behaviour be recovered from the prefix in a way that is useful for simulation, model checking or test derivation? That this question is far from trivial can be learned from studying [Esp94,Gra97]. In this paper we propose an answer by deriving a graph rewriting system from the complete finite prefix. We intend to use this graph rewriting system as the basis for constructing the complete unfolding, for simulation, for model checking both branching and linear time properties, and for test derivation.

The paper is structured as follows. After a short introduction to condition event structures in section 2, we adapt the definition of complete finite prefix in section 3. In section 4 we present a graph rewriting system model, and in section 5 we show how to derive a graph rewriting system from a complete finite prefix. Section 6 is for conclusions and further work. Appendix A addresses some correctness issues, and in Appendix B we show how the so-called graph transition system can play a role in model checking.

2 Condition event structures

In this section we define an event structure model which is very similar to a type of Petri nets called *occurrence* nets [NPW81,Eng91]; the role of places is taken by *conditions*, and there is a binary relation between conditions called *choice*. Condition event structures have been defined in [LB99] for modelling process algebra, and the choice relation is there to model the choice operator in process algebra. Removing the choice relation from the model yields a model equivalent to occurrence nets; in fact the results in this paper also hold for occurrence nets.

Definition 2.1 A *condition event structure* is a 4-tuple $\mathcal{E} = (D,\ E,\ \sharp,\ \prec)$ 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 ${}^\bullet x = \{y \in D \cup E \mid y \prec x\}$, the *postset* $x^\bullet$ is defined by $x^\bullet =$

$\{y \in D \cup E \mid x \prec y\}$. The *initial marking* M_0 is defined by $\{d \in D \mid {}^\bullet d = \emptyset\}$.

Definition 2.2 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 2.3 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: $d_1 \sharp d_2 \Rightarrow {}^\bullet d_1 = {}^\bullet d_2$ □

A well-formed condition event structure becomes a prime event structure [Win89] if we delete the conditions. Similarly as for prime event structures we define a *configuration* as a set of events C that is conflict-free ($\forall e, e' \in C : \neg(e \# e')$) and left-closed ($\forall e \in C : e' \leq e \Rightarrow e' \in C$).

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 $\mathcal{D}$, $\sharp(\mathcal{D}) = \{d' \mid \exists d \in \mathcal{D} : d \sharp d'\}$.

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 $e_1 \ldots e_n$ such that there are markings $M_1, \ldots, M_n$ with $M_0 \xrightarrow{e_1} M_1 \longrightarrow \ldots \xrightarrow{e_n} M_n$. It can be proven that $\{e_1, \ldots, e_n\}$ is a configuration if and only if $e_1 \ldots e_n$ is an event sequence.

Two nodes x and x' are said to be independent, notation $x \asymp x'$, iff $\neg(x \leq x') \wedge \neg(x' \leq x) \wedge \neg(x \# x')$. If X is a set of conditions, $x \asymp X$ iff $\forall x' \in X : x \asymp x'$.

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

Theorem 2.4 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 $Cut(Conf(M)) = M$. □

The condition event structure model can be used to represent the *unfolding* of a system. The unfolding of a system has been described as "a concurrent version of the usual notion of the unfolding of a loop" [Esp94]. An unfolding can be constructed from a Petri net. In that case we assume the conditions are labelled with places of the Petri net, and the events are labelled with

transitions. Algorithmically an unfolding can be constructed by starting with conditions labelled with the initial marking of the Petri net, and then repetetively adding new events and conditions for transitions that are enabled by this marking [ERV97]. For a precise definition in terms of net homomorphisms we refer to [Eng91].
A similar algorithm has been defined for process algebra expressions [LB99]; in that case the conditions are labelled with a kind of sequential components similar to the one in [Old91], and the events are labelled with action occurrences.

Example 2.5 Consider the process algebra expression $(a;\mathbf{stop} \mid b;\mathbf{stop}) + c;\mathbf{stop}$ where $\mid$ is the parallel operator without synchronization. Then the unfolding is given in figure 1. Conditions are indicated by circles and events by action names. The choice relation is represented by a dotted line, the flow relation by arrows. Note that here the unfolding is finite as there is no recursion in the process algebra expression; in general an unfolding may be infinite. In figure 1 there is also the Petri net corresponding to the process algebra expression, according to [Old91]. Suppose we look at the marking after transition a has happened. This marking is labelled with the set of components $\{\mathbf{stop} \mid, \quad \mid b;\mathbf{stop} + c;\mathbf{stop}\}$. Now this set of components does not directly correspond to the components of the expression $\mathbf{stop} \mid b;\mathbf{stop}$, which is the process algebra expression after a. This lack of correspondence is a nuisance when applying the complete finite prefix approach. For this reason we have defined the condition event model, with which it is possible to directly compute the unfolding, without making the detour via Petri nets and while avoiding the above problem. □

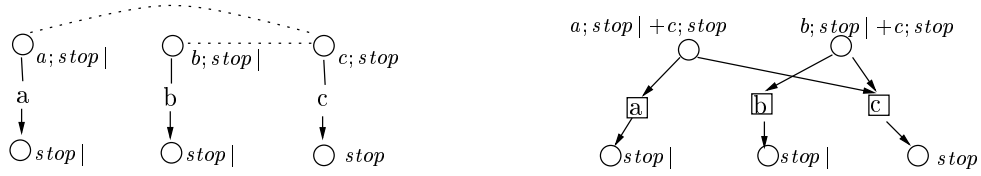

Fig. 1. Example of a process algebra unfolding and its corresponding Petri net

So we work with *labelled* condition event structures: condition labels are places in the case of Petri net unfoldings, and process algebra components in the case of process algebra unfoldings.
For a Petri net unfolding we can define a mapping St that maps a cut to a reachable marking of the Petri net (by taking the set of places that the conditions in the cut are mapped to). For a process algebra we can define a mapping St that maps a cut to a process algebra expression (the expression that can be decomposed into the components of the cut). In this paper both a reachable marking and a reachable process algebra expression will be called *a reachable state*. So: $St(M)$ is the reachable state of cut M, and it is defined as the set of labels of the conditions in M.

3 A complete finite prefix for condition event structures

We recapitulate the complete finite prefix approach as originally definied by McMillan for 1-safe Petri nets [McM92,McM95a,McM95b]. A *complete finite prefix* of an unfolding U is an initial part of the unfolding that has the following two properties:

- for each cut M of U there is a cut M' of the complete finite prefix such that $St(M) = St(M')$, so the prefix contains all reachable states;
- for each cut M with $M \xrightarrow{e}$ in U there is a cut M' in the prefix such that $M' \xrightarrow{e'}$ and e and e' are labelled with the same transition or action.

Let e be an event of a condition event structure, then the *local configuration* $[e]$ is defined by $[e] = \{e' \in E | e' \leq e\}$ (it is easy to prove that $[e]$ is indeed a configuration). It is convenient to assume a pseudo-event $\perp$ for which $[\perp] = \emptyset$. We assume we have a so-called *adequate ordering* $\sqsubset$ on the configurations of a condition event structure ; one of the properties of this ordering is that it is well-founded (see [ERV97] fo details). The original ordering defined by McMillan was $C_1 \sqsubset C_2 \Leftrightarrow |C_1| < |C_2|$; in [ERV97,LB99] optimizations have been defined that may lead to smaller prefixes.

Definition 3.1 Let U be an unfolding and let $\sqsubset$ be an adequate order on the configurations of U. An event e is a *cut-off event* if U has a local configuration $[e_0]$ such that $St(Cut([e])) = St(Cut([e_0]))$ and $[e_0] \sqsubset [e]$; e_0 is called the *corresponding* event of e. □

Definition 3.2 Let N be the set of nodes of unfolding $U = (D, E, \sharp, \prec)$ such that $n \in N$ iff no event causally preceding n is a cut-off event. Then $fp(U)$ is defined by $fp(U) = (D \cap N,\ E \cap N,\ \sharp \cap (N \times N),\ \prec \cap (N \times N))$ □

So $fp(U)$ contains all local configurations, and stops at cut-off events since their local configuration has been encountered already. The nice result proven by McMillan [McM95b]) is that this is enough to guarantee completeness, so $fp(U)$ contains also all non-local configurations and in fact is a complete finite prefix of U. In [McM95b] an algorithm is given that constructs this complete finite prefix directly from the Petri net. This algorithm is easily transformed into an algorithm that generates the complete finite prefix directly from a process algebra expression [LB99].
Note that in general an unfolding may have many different complete finite prefixes; we refer to $fp(U)$ as *the* complete finite prefix of U, or shortly as the finite prefix.

Example 3.3 Consider $B = P \mid_b Q$ with $P = a; b; P$ and $Q = c; b; (e; P + d; Q)$ where $\mid_b$ is a parallel operator with synchronization on b, similar to the LOTOS or CSP operator [BB87,Hoa85]. Then the finite prefix is given in figure 2; cut-off events are indicated by putting a box around them. □

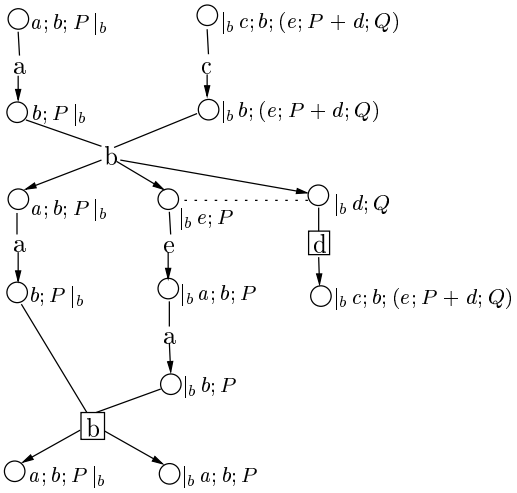

Fig. 2. Example of a complete finite prefix

Let M be a marking of a condition event structure $\mathcal{E} = (S, E, \sharp, \prec)$. Define the successor nodes of M by $N = \{x \in E \cup S \mid \exists y \in M : y \leq x\}$. Define $\Uparrow_{\mathcal{E}} M = (S \cap N,\ E \cap N,\ \sharp \cap (N \times N),\ \prec \cap (N \times N))$. It is easy to check that $\Uparrow_{\mathcal{E}} M$ is a well-formed condition event structure. We denote $\Uparrow_{\mathcal{E}} M$ by $\Uparrow M$ if $\mathcal{E}$ is an unfolding, and by $\Uparrow_p M$ if $\mathcal{E}$ is the finite prefix of an unfolding.
If C_1 and C_2 are two configurations of an unfolding Unf such that $St(C_1) = St(C_2)$, then $\Uparrow Cut(C_1)$ and $\Uparrow Cut(C_2)$ are isomorphic. So there is an isomorphism $I_{C_1}^{C_2}$ from $\Uparrow Cut(C_1)$ to $\Uparrow Cut(C_2)$. Let e be a cut-off event, with corresponding event e_0, and let $C_1 = [e_0]$ and $C_2 = [e]$, then we denote $I_{C_1}^{C_2}$ by I_e, and its inverse function $I_{C_2}^{C_1}$ by I_e^{-1}.

The function I_e^{-1} plays an important role in the proof of the completeness of the finite prefix, which roughly goes as follows: suppose we have some reachable state S in an unfolding, so there is a cut M in the unfolding with $St(M) = S$. Now either $Conf(M)$ does not contain a cut-off event, so M is also a cut of the finite prefix. Or $Conf(M)$ does contain some cut-off event e, but then there is also a cut $I_e^{-1}(M)$ in the prefix, with $St(I_e^{-1}(M)) = St(M)$, and $I_e^{-1}(M) \sqsubset M$. Repeat this procedure for $I_e^{-1}(M)$, since $\sqsubseteq$ is well-founded this can only be repeated a finite number of times, so eventually we arive at a cut in the finite prefix whose state is S.
So what this proof shows is: if M is a cut in the unfolding with $Conf(M)$ containing one or more cut-off events, then we can "shift back" M in the unfolding with the help of I_e^{-1} functions, until we have arrived at a cut M' with $St(M') = St(M)$ and where $Conf(M')$ does not contain any cut-off events (so M' is a cut in the finite prefix). We denote M' by $Shift(M)$; $Shift(M)$ is not necessarily uniquely determined (it may depend on the choice of the cut-off events over which the backwards shifting is performed) but that is not important for us here.

A complete finite prefix contains all reachable states and events; in a sense the

complete system behaviour is determined by the complete finite prefix. Can we recover this behaviour from the prefix, in other words: can we construct an unfolding from a complete finite prefix? This problem will be dealt with in the next two sections: we will transform the finite prefix into a graph rewriting system that we intend to use as the basis of simulation and model checking.

4 A graph rewriting system model

In this section we present a graph rewriting system model which has been inspired by the model in [QJ96], where a graph rewriting system formalism is used to generate infinite state transition systems.

Definition 4.1 A *graph* X is a tuple $(\mathcal{E}, \mathcal{C})$ with $\mathcal{E}$ a labelled condition event structure and $\mathcal{C}$ a set of events in $\mathcal{E}$ such that for each $e \in \mathcal{C} : e^{\bullet\bullet} = \emptyset$. We call $\mathcal{C}$ the set of *cut-off* events of X. □

When we talk about events, conditions, cuts etc. in a graph, we refer to the events, conditions, cuts etc. of the condition event structure of that graph. In a complete finite prefix each cut-off event has a corresponding event e_0, but that need not be the case for a cut-off event of an arbitrary graph. A cut-off event in a graph is simply an event that has been marked as such, and that has no causal successor events. However, as will be seen in the next section, graphs can be constructed from complete finite prefixes, and each cut-off in a graph results from a cut-off event in that complete finite prefix.

An event e can occur in different graphs. If we talk about the local configuration of e, we need to know what is the graph of this local configuration. Therefore we use the subscript of the graph in the notation of the local configuration, e.g. $[e]_X$ is the local configuration of e in graph X.

Definition 4.2 A graph rewriting system is a tuple (G, G_0) where G is a set of graphs, and $G_0 \in G$ is the initial graph, under the following constraint:
if e is a cut-off event in some graph $X \in G$, then there is a graph $Y \in G$ such that $St(Cut([e]_X)) = St(Cut([\bot]_Y))$. □

Recall that $[\bot]_Y$ means the empty configuration in graph Y. So the constraint says for each cut C corresponding to a cut-off event in some graph X there is another graph Y whose initial cut has the same state as C.
It is convenient, especially for graphical representations, to be able to refer to graphs via graph names. So we assume some injective mapping $name : G \rightarrow Names$ into a set $Names$ of graph names. We often will be sloppy about this and identify a graph with its name.

If e is a cut-off event in graph X, we call the cut $Cut([e]_X)$ an *instantiation* of graph Y if it has the same state as the initial cut of Y. In pictures we use the convention that we draw a black line through the initial cut of a graph (and put the graph name close to it), and we draw a gray line through an instantiation (and put the graph name of the instantiation close to it).

Example 4.3 An example of a graph rewriting system can be found in figure 3. This graph rewriting system corresponds to the complete finite prefix of figure 2, as will hopefully be clear after the next section. □

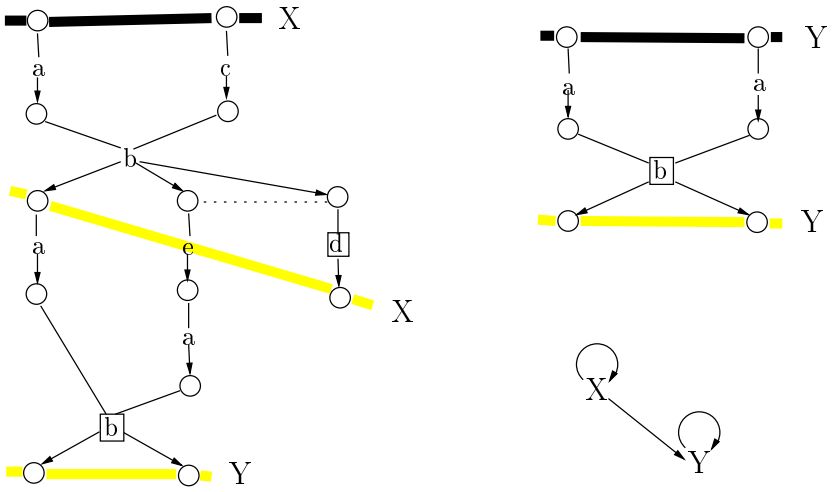

Fig. 3. Example of a graph rewriting system

In figure 3 there is also a transition system which is given by the next definition.

Definition 4.4 Let (G, G_0) be a graph rewriting system. Then the *graph transition system* is the transition system with a set of nodes G (actually the names of G), a transitions $X \to Y$ iff X contains an instantiation of Y, and G_0 the initial node. □

We expect graph transition systems to play an important role in model checking; we will come back to this in section 7.

So now we have a graph rewriting system; what can we do with it? One use would be to consider it actually as a rewriting system similar to the one in [QJ96] by looking at the graphs as production rules and the graph instantiations as nonterminals. Repetively substituting the graph instantations by the corresponding graphs would lead to possibly infinite condition event structures.

A difference with the approach in [QJ96] is that in our rewriting system it is possible that in a graph there are events and conditions causally dependent on conditions in a a graph instantiation; this means that in applying a graph as a rewriting rule possibly these events have to be identified with some events in the graph that replaces the instantiation. Look for example at graph X in figure 3. In graph X, after the instantiation of X there is an event labelled a; if the instantiation of X is replaced by X, then this a event needs to be identified with the a event just after the initial cut of X. Note that e.g. the cut-off event labelled b is not an event after the instantiation X as it is in conflict with the cut-off event just preceding the instantiation of X.

There are some technical details to be taken care of, but it is possible to use the graph rewriting system in such a way that it will produce a possibly infinite unfolding. However, this is not the direction we will take, as there is a more interesting way of looking at a graph rewriting system. We will consider each graph to be generalization of the concept of state: a graph is in fact a set of states, namely all the possible cuts or configurations in that graph. If we want to know in what state a system is, we need to know at what graph, and at what cut or configuration in that graph, the system is. So a state is a pair of a graph and a configuration, and we define transitions between states:

Definition 4.5 Let (G, G_0) be a graph rewriting system, then we define an initial state $(G_0, \emptyset)$ and transitions:

- $(X, C) \xrightarrow{e} (X, C \cup \{e\})$ iff
  - C does not contain any cut-off events from X
  - $C \cup \{e\}$ is conflict free
  - $C \cup \{e\}$ contains $[e]_X$
- $(X, C) \longrightarrow (Y, C \setminus [e]_X)$ iff $e \in C$ is a cut-off event in X with $Cut([e]_X)$ an instantiation of Y (this is an *empty* transition).

□

Example 4.6 Let us look at figure 3. Consider the second event labelled a (the one after the event labelled b) in X, and let us denote by a slight abuse of notation its local configuration by $\{a, c, b, a\}$. Then we have

$(X, \{a, c, b, a\}) \xrightarrow{d} (X, \{a, c, b, a, d\}) \longrightarrow (X, \{a, c, b, a, d\} \setminus [d]_X) = (X, \{a\})$. □

If (X, C) is the result of an empty transition as defined above, it need not always be the case that C is a configuration of X (when C contains successors of a cut-off event in X). However, we can absorp empty transitions by defining $(X, C) \xrightarrow{e} (Y, D)$ (where C is a configuration without cut-off events of X, and D is a configuration without cut-off events of Y), iff either $(X, C) \xrightarrow{e} (Y, D)$ (so $X = Y$) or $(X, C) \xrightarrow{e} (X_1, C_1) \longrightarrow \ldots \longrightarrow (Y, D)$. In this way we have defined event transitions for a graph rewriting system. In a completely standard way (see e.g. [Lan92]) event sequences and partial order transitions can be defined. These event transitions are important as the basis for an interactive graphical simulator: only an initial part of the rest of the system behaviour is shown to the user, and this initial part is updated in a "lazy" way as the system run proceeds. The advantage of such a simulator over a simulator based on interleaving semantics is that the user is not forced to resolve choices that are there only because of the interleaving semantics.

In the definitions of event transitions we made use of configurations, so we only need to refer to events and not to conditions. Therefore it is possible to remove all conditions in a graph rewriting system, provided we introduce some way of explicitly denotating graph instantiations. In this way we have obtained a finite representation of infinite event structures. Sofar we have not

paid any attention to the event identifiers, which form the subject of appendix A.

Note that we can consider an "ordinary" interleaving transition system (e.g. the standard semantics for a process algebra expression) as a special case of a graph rewriting system, by considering each state as a graph, and each transition as a cut-off event.

5 Deriving a graph rewriting system from a finite prefix

How to construct a graph rewriting system from the finite prefix? The initial graph is easy: this is simply the finite prefix with its cut-off events. For each cut-off event e in the finite prefix we create a graph $\Uparrow_p Cut([e_0])$, where the cut-off events in this graph are those events marked cut-off in the finite prefix. How about the graphs corresponding to these cut-off events? If for a cut-off event e' its local marking is completely contained in the graph, then there is already a graph corresponding to it, namely $\Uparrow_p Cut([e'_0])$. However, this does not need to be the case. We can have a situation like in figure 4, where a graph X with a cut-off event e' are schematically represented. Suppose M is the initial cut of X in the finite prefix; $Cut([e'])$ is not completely in X. We now want to create a graph for e', where the instantiation of this graph is the cut along the dotted lines. If we call this cut M', then is not hard to see that $M' = Cut(Conf(M) \cup [e'])$.

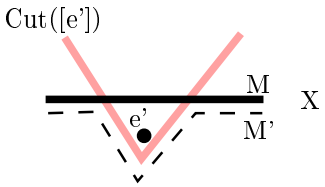

Fig. 4. A cut-off event inside a graph

So we are looking for a cut M'_0 for which $St(M'_0) = St(M')$ and that is "earlier" than M' (w.r.t. the adequate order $\sqsubset$). Note that M' is a cut in $\Uparrow_p Cut([e'])$, so there is an "earlier" cut with the same state in $\Uparrow Cut([e'_0]$, and this cut is given by $I_{e'}^{-1}(M')$.
An example of such a situation can be found in figure 5. Suppose we have created the graph for e, then $St(M) = St(Cut([e_0])) = \{3, 4\}$. Check that $St(M') = St(Cut([e_0] \cup [e'])) = \{2, 3\}$, and for $M'_0 = I_{e'}^{-1}(M')$ we have $St(M'_0) = \{2, 3\}$.
Now there is only one thing to take care of: it may be the case that M'_0 is not present in the finite prefix. This may happen if there is a cut-off event e'' in $Conf(M'_0)$ (such an event has been baptized a *tricky event* in [Gra97]). The remedy is simply that we then have to shift M'_0 backwards until it is in the finite prefix, i.e. we have to take $Shift(I_e^{-1}(M'))$.

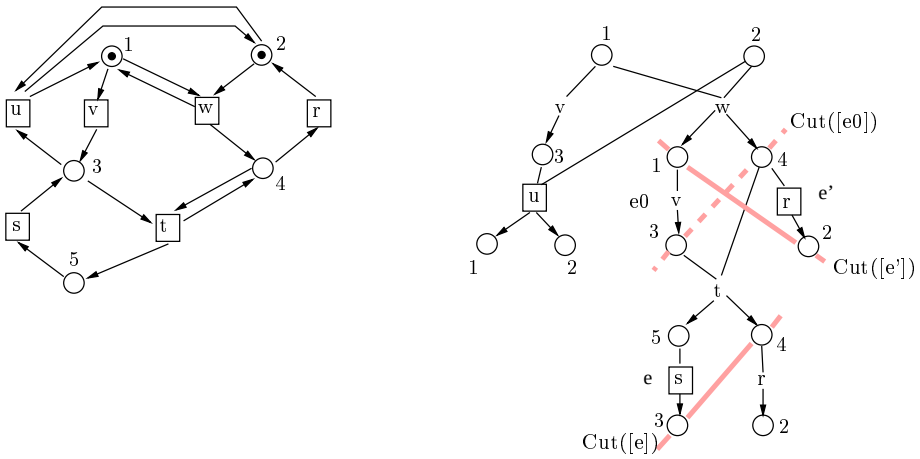

Fig. 5. A Petri net and its finite prefix

We have now explained all steps of Algorithm 5.1, which takes a finite prefix and returns a graph rewriting system. In this algorithm P, $\mathcal{C}$, G and G_0 are global variables; all other variables are local to the (recursive) procedure *Inside_graph*(M), which creates possibly new graphs for the cut-off events inside a graph with initial cut M.

Algorithm 5.1
Input: a finite prefix P with set of cut-off events $\mathcal{C}$
Output: a graph rewriting system (G, G_0)

$G_0 := (P, \mathcal{C})$;
$G := \{G_0\}$;
Inside_graph$(Cut([\bot]))$

where

process *Inside_graph*(M)
begin
forall cut-off events e in $\Uparrow_p M$ **do**
  $M' := Cut(Conf(M) \cup [e])$;
  if not $\exists\, Y \in G : St(Cut([\bot]_Y)) = St(M')$
  then
    $M'_0 := Shift(I_e^{-1}(M'))$;
    $\mathcal{C}' :=$ cut-off events in $\Uparrow_p M'_0$;
    $G := G \cup \{(\Uparrow_p M'_0, \mathcal{C}')\}$;
    Inside_graph(M'_0)
end

□

Example 5.2 Algorithm 5.1 transforms the finite prefix of figure 2 into the graph rewriting system of figure 3, and the finite prefix of figure 5 results in the graph rewriting system of figure 6. The interested reader will find that working out the algorithm for these cases greatly enhances the intuition for

complete finite prefixes and their corresponding graph rewriting systems. □

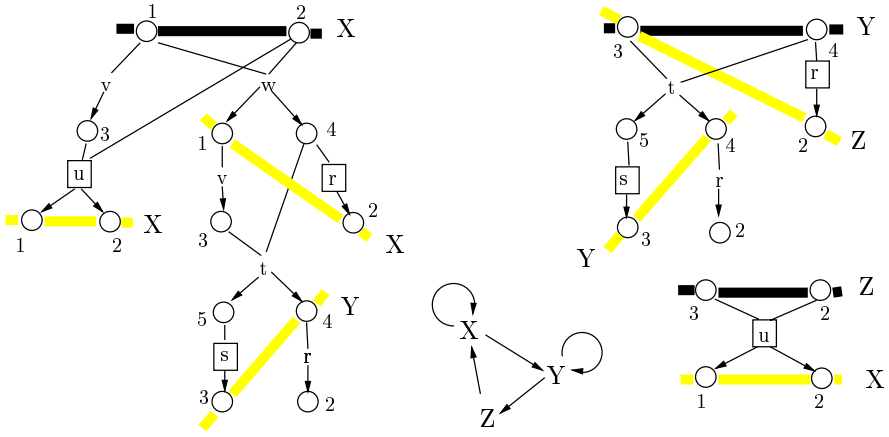

Fig. 6. Graph rewriting system corresponding to figure 5

In some cases it is possible to also create an instantiation at $Cut([e_0])$ for a graph that is instantiated at cut-off event e, and remove all events causally dependent on e_0; this would be an optimization, and there are several other possible optimizations, which are a topic of future work.

6 Conclusions

We have used the model of condition event structures [LB99] in which we can express unfoldings of either Petri nets or process algebra expressions. For these unfoldings a complete finite prefix according to McMillan can be defined [McM95a,ERV97,LB99].
We have defined a graph rewriting system model that can be used for producing the unfolding. More interestingly, this graph rewriting system can be seen as a generalisation of a transition system and event sequences can be derived via the definition of event transitions, which may form the basis of an interactive graphical simulator. We have presented an algorithm for transforming a finite prefix into a graph rewriting system, and we have defined a *graph transition system*.

In appendix A we have indicated how by parameterization of the graph rewriting system we can take care of the issue of event identifiers that forms an important aspect of the correctness proof. Finally in appendix B we have hinted at how to use the graph transition system as the underlying model for test derivation and model checking (for both branching and linear time properties).
The graph transition system may yield a very compact representation of the system behaviour. However, it is possible to find worst case examples in which the size of the graph rewriting system is exponential in the size of the complete finite prefix. We are currently studying several optimizations,

both for obtaining a smaller prefix (which may not be anymore complete, but still produces the complete unfolding via the graph rewriting system) and for avoiding the explosion in the size of the graph rewriting system, that will be the subject of a forthcoming paper.

Acknowledgements

Thanks to Ed Brinksma, Joost-Pieter Katoen, Diego Latella, Mieke Massink, Frank Wallner, and especially Theo Ruys for discussions and 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.

[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.

[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. In *Proceedings of CAV'99*, 1999. Accepted for publication.

[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.

[QJ96] Y.-M. Quemener and T. Jeron. Finitely representing infinite reachability graphs of cfsms with graph grammars. Publication Interne 994, IRISA, Rennes, France, March 1996.

[UK97] A. Ulrich and H. Koenig. Specification-based testing of concurrent systems. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification*, pages 7–22, 1997.

[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*, volume 354 of *Lecture Notes in Computer Science*, pages 364–397. Springer-Verlag, 1989.

Appendix A: event identifiers and correctness issues

Until now we have been rather vague about the labels of the events in condition event structures, prefixes and graph rewriting systems. These event identifiers however play an important role in proving the correctness of the approach. In this section we will sketch the approach for complete finite prefixes generated from process algebra expressions; we conjecture it is possible (and interesting) to adapt this approach for Petri nets.

Different event identifiers model different occurrences of actions. We have shown in [Lan92,LB99] how these event identifiers can be generated by having annotations of actions and a slight modification of the standard SOS rules. We assume that each occurrence of an action in a process algebra expression is indexed by a unique *action index* and each process instantiation by a unique *process identifier*. Here we assume action identifiers to be integers and process identifiers to be greek letters. The modified SOS rules yield transitions of the form $\xrightarrow{a,e}$ which stands for a transition labelled with action a and event identifier e; such an event is also denoted by a_e.

Example 6.1 Let P_α be a process expression where $P = a_1; c_3; P_\phi$. Then with the modified SOS rules in [LB99] we can derive the following sequence of transitions:
$P_\alpha \xrightarrow{a,\alpha 1} \xrightarrow{c,\alpha 3} P_{\alpha\phi} \xrightarrow{a,\alpha\phi 1} \xrightarrow{c,\alpha\phi 3} P_{\alpha\phi\phi} \ldots$.
From the expression $P_\alpha \mid_c Q_\beta$ where $Q = b_2; c_4; Q_\psi$ we have (after transitions with actions a and b) transitions $\xrightarrow{c,\alpha 3}$ and $\xrightarrow{c,\beta 4}$ synchronize to form a transition $\xrightarrow{c,(\alpha 3,\beta 4)}$. □

When defining the cut-off events of the finite prefix we consider as the state equality criterium (in e.g. $St([e]) = St([e_0])$) equality of the process algebra expressions stripped from all process indices. This leads for the expression $P_\alpha \mid_c Q_\beta$ of the previous example to the complete finite prefix in figure 7.

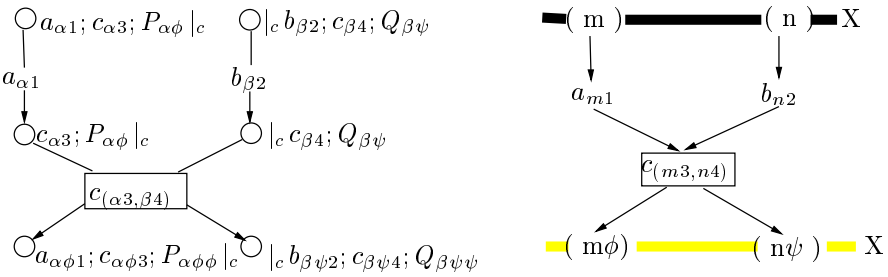

Fig. 7. Example of parameterization

We want the graph rewriting system that is generated from a complete finite prefix to produce the same event identifiers as the original process algebra expression. In order to achieve this we have to parametrize the events and conditions in a graph. The parameters are introduced at the conditions of the initial cut of a graph. In figure 7 we see the graph corresponding to

the prefix in the same figure (the conditions inside that graph have been deleted, so this graph rewriting system generates a prime event structure). The parameterization can be obtained by replacing all process indices in a consistent way by parameters, and introducing at a condition in the initial cut of a graph the parameter corresponding to the process index of the action prefix at that condition.
The initial graph of a graph rewriting system needs to be instantiated with the appropriate process indices. For instance, the graph rewriting system corresponding to the complete finite prefix in figure 7 is $(\{X\}, X(\alpha, \beta))$.

Theorem 6.2 Let B be a process algebra, and (G, G_0) the graph rewriting system generated from the complete finite prefix of B. Then: σ is an event sequence of $B \Leftrightarrow \sigma$ is an event sequence of the unfolding of $B \Leftrightarrow \sigma$ is an event sequence of (G, G_0). □

Note that the involvement with event identities is mainly important for the proof of the correctness of the approach; in applying this theory one is often interested in just the action labels, and need not be bothered by event identifiers.

Appendix B: Graph transition systems and model checking

In section 4 we have defined for a graph rewriting system a graph transition system, which has as nodes the graphs of a graph rewriting system, and transitions $X \rightarrow Y$ iff Y is an instantiation in graph X. We can label such a transition with a set of labelled partial orders: if the instantiation of Y is reached from the initial cut of X via a labelled partial order p, p is in the set that labels the transition $X \rightarrow Y$. Note that it is possible that there are different instantiations of Y in X, hence we need a set of partial orders (see the example in figure 6 where there are different instantiations of X in graph X).
We think that looking at the graph transition system gives an insight in the structure of a system that is difficult to obtain from just looking at the complete finite prefix. We expect that in addition the graph transition system might play an important role in validating a system, for instance as the basis for model checking or test derivation.

One possibility would be to check branching time logics properties in the spirit of [Esp94]. Let us look at a simple example to get the basic idea. Suppose for the graph rewriting system in figure 6 we want to check whether state $\{4, 5\}$ is always reachable from state $\{2, 3\}$. Now as a first step we have to know where in the graph rewriting system these states are. We find e.g. that $\{2, 3\}$ is in a partial order from X to X, and $\{4, 5\}$ is in a partial order from Y to Y. When all information of this kind is added to the graph transition system we obtain the system in figure 8. The transition system in this figure can

be subjected to standard branching time model checking algorithms based on labelling states.

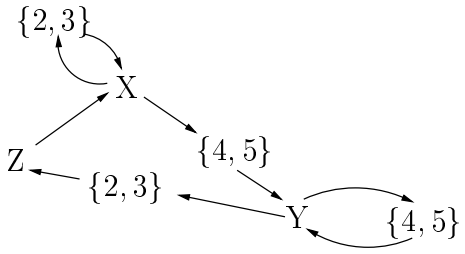

Fig. 8. State information added to a graph transition system

Another approach could be to check linear time temporal properties in the style of [Wal98]. Then the approach could roughly be the following. First translate a formula ϕ into an automaton $A_{\neg\phi}$. Now take the synchronization of $A_{\neg\phi}$ and the system where we only synchronize on the so-called *visible* actions (which means we restrict ourselves to stutter-invariant properties, not including the *next* operator of linear time logic). From this synchronization we construct the complete finite prefix and subsequently the graph transition system. We mark a transition if it is labelled with a partial order containing an accepting state or transition; now we can use standard algorithms for detecting the presence of a cycle with a marked transition.

Another use of the graph transition system would be to use it as the basis for test derivation by adapting standard transition tour algorithms in a similar way as has been done in [UK97] for a slightly different model.

The benefit of using the graph transition system is that it is a reduced and compact transition system. We expect this approach to offer a better performance (w.r.t. standard interleaving methods) for systems in which there is a high level of concurrency (e.g. resulting from the existence of parallel subsystems), as in such systems the interleaving transition system will be of exponential size; ofcourse this expectation needs to be confirmed by experimental results that will be worked at as soon as the implementation of the approach (that is currently being worked on at the University of Twente) has been finished.