Application of Partial-Order Methods to Reactive Programs with Event Memorization

Frédéric Herbreteau, Franck Cassez, Olivier Roux*
IRCyN/CNRS UMR 6597
1 rue de la Noë, BP 92101
44321 Nantes cedex 03, France

September 25, 2001

Abstract

We are concerned in this paper with the verification of reactive systems with event memorization. The reactive systems are specified with an asynchronous reactive language Electre the main feature of which is the capability of memorizing occurrences of events in order to process them later. This memory capability is quite interesting for specifying reactive systems but leads to a verification model with a dramatically large number of states (due to the stored occurrences of events). In this paper, we show that *partial-order* methods can be applied successfuly for verification purposes on our model of reactive programs with event memorization.

The main points of our work are two-fold: (1) we show that the *independence relation* which is a key point for applying partial-order methods can be extracted automatically from an Electre program; (2) the partial-order technique turns out to be very efficient and may lead to a drastic reduction in the number of states of the model as demonstrated by a real-life industrial case study.

Keywords: transition systems, reactive languages, composition, partial-order methods.

Contents

*e-mail: {Frederic.Herbreteau | Franck.Cassez | Olivier.Roux}@ircyn.ec-nantes.fr

1 Introduction

The purpose of this paper is to *produce a reduced model* for the verification of reactive systems [BB91, MP93, Pnu86b, PH85] with event memorization, by the means of *partial-order methods* [Ove81, Val91b, GW91, Pel93, HP94, God96a].

Framework. The problem originated in the definition of a semantic model of an *asynchronous reactive language*: Electre [PRH92, CR95]. Indeed, this language provides the user with the ability of specifying that an event can be stored in order to be processed later (if it must not be processed when it occurs). However, the transition system which gives the semantics of an Electre program has a number of states that grows exponentially with the number of the so-called *memorizable* events. For verification purposes, this state explosion problem prevents the user from using standard model-checking techniques [McM93, GL94].

Partial-Order Methods. Recently, *partial-order methods* [Ove81, Val91b, GW91, Pel93, HP94, God96a] have proved to be useful and efficient in reducing the explored state-space in a verification phase. These techniques rely upon the equivalence between interleavings of actions of a transition system. The structure of the model for the memorizing device is such that each one of its state can also be seen as the interleaving leading from the initial state towards this state. In this paper, we show how partial-order methods can be successfully applied to the verification of Electre programs leading to the production of reduced models for the memorizing device. Particularly, we introduce an efficient method for the computation of the dependence relation directly from Electre programs.

Outline of the Paper. We first introduce, in section 2, the specification language Electre, and the semantic model on which verification can be carried out. Then, section 3 gives a quick overview of the key concepts of partial-order methods that will be used in section 4. This section is the core of the paper: we give the method and algorithms for applying partial-order methods to Electre programs. Section 4 also figures out complexity issues and results that illustrate the performance of the method (particularly, an industrial case study is exhibited). Eventually, in section 5, we conclude and give some directions for future work.

2 Electre: an Asynchronous Reactive Language with Event Memorization

In this section, we sketchily introduce the Electre language and its semantic model. For a detailed presentation the reader is referred to [CR95, SFRC99]. This language has been used for the modelization and the verification of several problems. In particular, [BBRR97] and [BR99] presents some recent results concerning temporal verification associated to it.

The main point of this section is the event memorization since it is the most original feature of the language and, moreover, we will focus on it in Section 4.

2.1 Asynchronous Features

The Electre language is based on an asynchronous assumption which is better described by the definition of its fundamental components:

modules which are lasting actions, of finite but non-null duration (they can be preempted and later resumed),

events which gather the occurrences of signals and can be stored; two occurrences are considered to occur at distinct instants in a continuous time space.

With these basic features are associated:

properties of modules or events; modules can be non-preemptible, preemptible, resumed from the last reached point or restarted from the initial point; events can be fleeting events or stored ones,

operators to combine modules and events: sequence, loop, parallel structures can be built with modules; starting or preempting a module by an event. Various sorts of preempting structures can be used: exclusive or parallel.

Module-based structures (a parallel composition of several sequences of modules for instance) and event-based structures (e.g., a preemption structure with an exclusive composition of parallel preemption structures) can be built and mixed according to the syntactic rules.

In Electre, the algorithmic content of a module is abstracted away and is supposed to be some executable code written in any sequential language. Thus, an Electre program only refers to module or event identifiers and describes the temporal behavior of the system as operations on its modules: starting and preempting with respect to the occurrences of events (which can be hardware or software originated) and the special events end_M for all its modules M.

The asynchronous features of the Electre language are in the language basic components: modules and events (as defined above) allow the developer to specify the asynchronous nature of its application. Moreover one of the asynchronous features of Electre is precisely its ability to memorize events in order to process them later. Beside this, the internal treatment of the language (compilation, execution, ...) is quite similar to synchronous languages [LBBG86, HCRP91, Bd91].

2.2 Partial Semantics of Electre Programs

In this subsection, we introduce the semantics of the Electre language without taking into account the memorization aspects. The model of an Electre program is a transition system that we call the *control system* $\mathcal{C}$.

2.2.1 Syntax

The Electre abstract syntax is based on two sets: module identifiers Mid on the one hand, and event identifiers Eid on the other hand. We also partition the set of events into two disjoint sets: $E = E_{@} \cup E_M$ where $E_{@}$ is the set of fleeting events and E_M the set of memorizable events. There are two types of components : *module structures* MS and *events structures* ES which are inductively defined as follows[1]:

- $nil \in \mathsf{NIL}$,
- $m \in \mathsf{Mid} \Longrightarrow m \in \mathsf{MS}$,

[1]for semantical reasons we add a special entity *nil* of type NIL which stands for "nothing".

- $e \in$ Eid $\Longrightarrow$ e, $@e \in$ ES (one-time storage event and fleeting event),
- $P, P' \in$ MS $\Longrightarrow$
  - loop P end loop $\in$ MS (repetition),
  - $P \parallel P' \in$ MS (parallel module structure),
  - $P\,;P' \in$ MS (sequential module structure),
- $E, E' \in$ ES $\Longrightarrow$
  - $E \parallel E' \in$ ES (parallel event structure),
  - E or $E' \in$ ES (exclusive event structure),
- $P \in$ MS, $E \in$ ES $\Longrightarrow$
  - P await $E \in$ MS (preemption),
  - E launch $P \in$ ES (launching).

An Electre program is an element of {MS, NIL}: we consider hereafter that Electre programs are represented by their parse trees with nodes decorated with the type of the components (MS or ES or NIL).

2.2.2 Semantics

According to the semantics of the language [CR95], for any Electre program P, and any event e, we can determine the program P' which is obtained after *taking into account* the occurrence of event e (an example of such a calculus is given in Figure 2).

The formal semantics of the complete Electre language is given in [CR95]. Here we give in Appendix A the semantics for the subset of the language we will need in this paper. Judgements are of the form (these are conditional rewriting rules written in a SOS style):

$$\frac{\mathcal{E} \vdash P_i : \tau \rightarrow P'_i : \tau' \qquad i \in [1, n]}{\mathcal{E} \vdash op(P_1, \ldots, P_n) : \theta \rightarrow op'(P'_1, \ldots, P'_n) : \theta'}$$

The rule reads as : if the P_i's of type τ rewrite in P'_i of type τ' in the environment $\mathcal{E}$, then a program built from the P_i's with the operator $op(P_1, \ldots, P_n)$ rewrites in $op'(P'_1, \ldots, P'_n)$. $\mathcal{E}$ is the environment and θ and θ' are types in {MS, ES, NIL}. By convention: $p \parallel nil = nil \parallel p = p$, and $p; nil = nil; p = p$ and, nil await E, is written await E.

We then define a *reaction* of the system in the state s to the event e by the transition relation:

$$s \xrightarrow{e} s' \qquad \textit{iff} \qquad \{e\} \vdash s : \{\mathsf{MS}, \mathsf{NIL}\} \rightarrow s' : \{\mathsf{MS}, \mathsf{NIL}\}$$

Thus, we obtain a finite transition system; each one of its states is featured by the program that has to be executed at the current time. And, since the Electre language is an asynchronous reactive language, a transition is labeled with a *single* event's name. The set of reactions from the initial program yields to the *control transition system* associated to each Electre program.

As a running example, we take the program of Figure 1. The semantics of the operators

```
PROGRAM Foo ;
loop
        await
             { e1 launch A || e2 launch B };
end loop ;
END Foo ;
```

Figure 1: An Electre program.

are intuitively given by:

- the *loop–end loop* structure stands for a never-ending loop of the inner code,
- the *await* operator is both applied to an event structure (within braces in the example) and to a module structure as in P `await` E: this structure consists in waiting for an event of E which preempts the execution of the modules in P.
- an *event structure* can be built with a parallel operator (||) meaning that its components are being concurrently waited for; as soon as one of them has occurred, the parallel event structure imposes that the other part is being waited for. The exclusive operator *or* can also be used to define event structure: the meaning of which is that the first occurrence of the events appearing in the structure starts the event structure and the others are no more being waited for,
- the components of an event structure are made of events, modules and the "`launch`" operator; the fragment e_1 `launch` A means that the occurrence of e_1 launches module A.

Some reactions corresponding to the first steps of the execution of program Foo's `loop` structure inner code are depicted in Figure 2 (the next steps which are not figured here would be to wait for end_A and end_B, the events that correspond to the end of these modules).

The control transition system associated with an Electre program describes the effects of the processing of the events' occurrences in all the possible states. This transition system does not describe memorization and batch processing of events.

2.3 Semantics of Memorization

As we have already mentioned, there are two kinds of events: the *fleeting* ones ($E_@$), which have an effect on the execution only when they are being waited for, since their occurrences

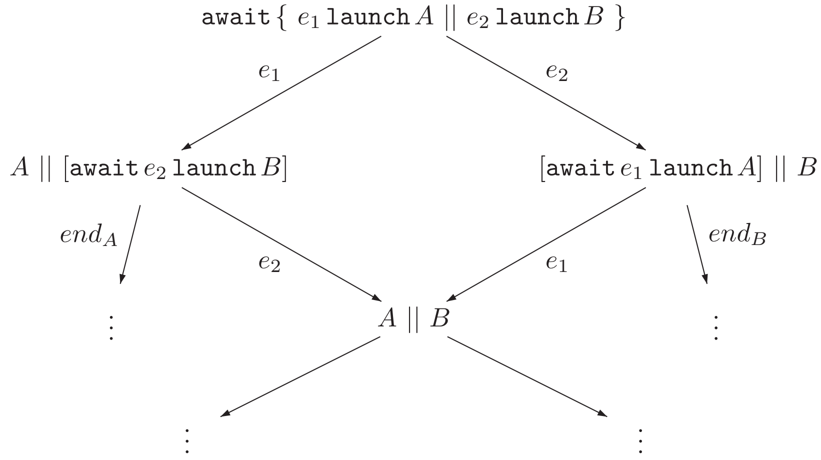


Figure 2: First transitions of the `loop` structure body of program Foo.

are never stored; and the *memorizable* ones (E_M), which may have a postponed effect, since they are stored (at most once) if they occur when they are not being waited for. Thus, we have a partition of the set of events: $E = E_@ \cup E_M$ (and $E_@ \cap E_M = \emptyset$).

Fleeting Events. The fact that the occurrence of an event is to be stored is an information given by the *state* of the control transition system. A fleeting event can always be taken into account immediately:

- it may bring about a change in the control system (launching a module execution, interrupting some module executions, ...),
- or it may have no effect (since it brings about no change), which indeed means the occurrence is taken into account (but its effect on the system is null).

Consequently, all the states of the control transition system are source states for transitions labeled by fleeting event: in every state, a fleeting event can be taken into account.

Memorizable Events. Memorized occurrences of events must be processed as soon as possible and with priority to the oldest memorized occurrence in case of conflict. The priority will be dealt with in the sequel. A memorized occurrence of an event can be taken into account in each state where a spontaneous occurrence of the same event can be taken into account. Moreover, the effect of taking into account a memorized occurrence of an event is the same as taking into account a spontaneous occurrence of the same event.

Event Memorization. The complete semantics of the language (dealing with the event memorization) imposes to add a structure for storing memorized occurrences. Since only one occurrence of event is stored, we could include this memorization in the control state of the transition system: the drawback of this technique is that it may bring about a state explosion in the number of control states. This is why we choose to store the occurrences of events in a separate list. We use a so-called *FIFFO*[2]*-transition system* which is a transition system with a kind of FIFO-list manipulated in a special way, which we will describe hereafter. In this FIFFO-transition system, we define the transition relation by:

1. all the transitions of $\mathcal{C}$ are transitions of $\mathcal{F}$,
2. with every transition $s \xrightarrow{e} s'$ of $\mathcal{C}$ is associated a transition $s \xrightarrow{\ominus e} s'$ in the FIFFO-transition system $\mathcal{F}$; this transition corresponds to taking into account a memorized occurrence of e (note that the source and target state of such a transition are the same as for the transition labeled e: this means that the effect of taking into account the stored occurrence of e is exactly the same as taking into account the occurrence of e),
3. for every state s of the control transition system, for each memorizable event which is not the label of a transition of which s is the source, we add in the FIFFO-transition system $\mathcal{F}$ a transition $s \xrightarrow{\oplus e} s$ (leading to the same state, but simultaneously recording the occurrence).

Synchronization of the FIFFO-Transition System and the FIFFO-List. To reduce the explored state-space, we can isolate the FIFFO component which is the cause of the state explosion. Thus, we synchronize the reduced list (see Section 3) with the control system.

First, we define the automaton for the FIFFO-list: this is straightforward and an example of an automaton for a list with two events e_1 and e_2 is given in Figure 4, Page 17. Transitions labeled $+e_i$ (resp. $-e_i$) correspond to storing (resp. removing) event e_i in the FIFFO-list. One can point out that from state e_1e_2 a transition with label $-e_2$ exists although e_1 is the oldest item of the list; this is the result of the FIFFO-list management where an event is taken into account as soon as possible: if e_1 cannot be processed and e_2 can, then e_2 is dequeued. Note also the τ transition which allows the FIFFO-list to remain idle.

The synchronization of the automaton of the FIFFO-list and the FIFFO-transition system is defined by the following rules:

- if transitions labeled $\ominus e_{i_1}, \ominus e_{i_2}, \ldots, \ominus e_{i_k}$ are enabled in the FIFFO-transition system, and one of these events is the oldest occurrence in the FIFFO-list, namely e_{i_l}, then transitions labeled $\ominus e_{i_l}$ and $-e_{i_l}$ are synchronized in the two transition systems,
- finally, if no $\ominus e / - e$ can be synchronized,
  - transitions labeled e in the FIFFO-transition system are synchronized with τ in the FIFFO-list,

[2]First In First Fireable Out.

– transitions labeled $\oplus e$ in the FIFFO-transition system are synchronized with transitions labeled $+e$ in the FIFFO-list.

Remarks. Priority is given to the processing of stored events. Moreover the management of the FIFFO-list is such that the stored events are taken into account as soon as possible and in case of conflict between stored occurrences with priority to the oldest. An execution may contain successive $\ominus e$ (resp. $-e$) transitions.

3 Independence Relation and Partial-Order Methods

Using transition systems for modelization purpose often leads to the so-called "state-space explosion problem". However, an important feature of a lot of modelled systems (e.g. parallel programs, communication protocols) is the built-in compositional nature. It is expressed by the *sometimes irrelevant order* between some of the system actions: from a given state of the transition system, whatever the order between some actions is, the reached state is always the same one. Of course, for the verification of some kinds of properties, we can get rid of this order since it is not useful (e.g., liveness, deadlocks detection). But, the order remains meaningful in many other cases.

Interleavings are not the panacea. Thus one can distinguish a lot of other "cheapest" models for concurrency: *partial-order models* (e.g., [Lam78, Maz86, Pra86, Win86]), *partial-order temporal logics* (e.g., [PW84, KP86, KP87, Pen88, Pen90]), among others.

On the other hand, for verification purposes, one aims at reducing the number of states that must be visited and/or the number of transitions that must be explored. Several techniques have been developped in order to take advantage of the concurrency for state-space reduction, particularly: *virtual coarsening of atomic actions* ([Pnu86a]), *nets unfoldings* (e.g., [McM92, Esp94]), methods by Overman ([Ove81]), strategies for property-proving without considering all interleavings (e.g., [AFdR80, EF82, SdR89, KP92b, JZ93]) and heuristic selection of interleavings (e.g., [GH85, Wes86, Hol87]).

More precisely, partial-order methods have been developped for verification purposes. They are intended to reduce the state-space by exploring only one of the interleavings of *independent events* (see Definition 1). They first appeared in [Val88a, Val88b] and independently in [God91, GW92]. These methods have met a great success, and consequently have been improved and adapted in a lot of frameworks [Val91a, GW91, HGP92, GP93, Pel93, Val93, GW93, HP94, Pel94, GHP95, God96a, Pag96, GKPP99]. One can consider [God96b] as a reference on partial-order methods, as it provides a description and comparison of these methods. All of them rely upon the *independence* relation between the actions of a given transition system.

3.1 Dependence and Independence between the Actions of an Automaton

Partial-order methods rely upon commutative transitions in sequences of transitions. Such transitions are called *independent* transitions, because they neither enable nor disable each other, so the order they appear in is irrelevant.

The following definition, adapted from [KP92a], gives the conditions of independence between two actions of an automaton. Then, the interleavings on independent actions are considered to be equivalent.

Definition 1 *Two actions a_1 and a_2 of an automaton are* independent *if the following two conditions are true (otherwise they are said to be* dependent*) :*

1. *if a_1 (a_2) is enabled in a state s and $s \xrightarrow{a_1} s'$ ($s \xrightarrow{a_2} s'$), then a_2 (a_1) is enabled in s iff a_2 (a_1) is enabled in s' (independent transitions can neither disable nor enable each other),*

2. *if a_1 and a_2 are enabled in s, then there is a unique state s' such that both[3] $s \stackrel{a_1 a_2}{\Longrightarrow} s'$ and $s \stackrel{a_2 a_1}{\Longrightarrow} s'$ (commutativity of enabled independent transitions).*

As we focus on interleavings of actions, and particularly, as far as we will be interested in the equivalence of interleavings of independent actions, we will use a formalism in order to represent them. Such a representation comes from the *trace* theory of Mazurkiewicz [Maz86]. A *trace* should be seen as a kind of equivalence class on interleavings of independent transitions.

Definition 2 *Let w be a sequence of transitions $t_1 t_2 \ldots t_i t_{i+1} \ldots t_n$. A trace $[w]$ represents all the sequences of transitions obtained from w by successively permuting adjacent independent transitions: $t_1 t_2 \ldots t_{i+1} t_i \ldots t_n$.*

Finally, the partial-order methods rely upon the following property that states the equivalence of behavior between sequences belonging to the same trace [GW93]:

Property 1 *If $s \stackrel{w_1}{\Longrightarrow} s_1$, $s \stackrel{w_2}{\Longrightarrow} s_2$ and $[w_1] = [w_2]$, then $s_1 = s_2$.*

This property justifies all the reductions that can be done by the way of partial-order methods.

3.2 Partial-Order Methods

The aim of partial-order methods is to optimize the amount of states and/or transitions explored, particularly with verification purposes in mind. Sometimes, people prefer to generate a reduced model and then model-check it (e.g., [Val91b, Pel93]), others use partial-order methods during an "on-the-fly" verification (e.g., [GW91, Val93, Pel94, GPS96, WW97]).

[3] We write $s \stackrel{w}{\Longrightarrow} s'$ to mean that the sequence of transitions w leads from s to s'.

```
Stack.push(s_0); H := ∅
while Stack ≠ ∅ do
  s := Stack.pop()
  if s ∉ H then
    H := H ∪ {s}
    for each t ∈ enabled(s) do
      s' := succ_t(s)
      Stack.push(s')
    end for
  end if
end while
```

Algorithm 1: Classical depth-first exploration of a state-space.

But, whatever the purpose partial-order methods are used for, one needs to differentiate the enabled transitions of a given state: on the one hand, the dependent ones, and on the other hand the independent ones. To reduce the number of transitions which have to be visited from a state s, one only selects and explores a *sufficient* subset of them. As a matter of fact, there mainly exists two methods[4] in order to do that: the *persistent-sets* method [Ove81, Val91b, GW92, GP93, God96a], and the *sleep-sets* method [God91, God96b].

Both methods are introduced using the framework of state-space exploration as it is a very general one (it is used during "on-the-fly" verification, as well as for reduced state-space generation). Algorithm 1 pictures a classical depth-first state-space exploration.

3.2.1 Persistent-Sets Method

In this first method, the selected subset only contains transitions which are dependent (independent ones will remain enabled in the reached states). Such a subset is called a *persistent-set* as it is not affected by transitions which are outside of it.

Persistent-Set. We define a set T of enabled transitions in s as a *persistent-set* whenever their occurrences cannot be affected by execution of transitions not in T, from s. Here is a definition of persistent-sets from [GW93]:

Definition 3 *A set T of transitions enabled in a state s is* persistent *in s if and only if, for all transitions $t \notin T$ such that there exists a sequence:*

$$s = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \cdots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n = t} s_{n+1}$$

leading from s to t and including only transitions $t_i \notin T$, t is independent (see Definition 1) with respect to all transitions in T.

[4] One can consider the *stubborn-sets* method [Val88a, Val88b, Val91b, Val93] as a persistent-sets one. Indeed Godefroid proved that stubborn-sets are also persistent-sets in [God96b].
There also exists some other methods like *faithful decompositions* [KP92b] or *ample-sets* [Pel93] that are quite similar to the persistent-sets one.

Notice that, the set of all enabled transitions in a state s is always persistent.

Computation. The computation of a persistent-set is performed on the static structure of the considered transition system: it does not rely on the way the state-space exploration is done. The following algorithm [Pag96] allows one to compute a persistent-set EP associated with the reached state s.

1. EP is initialized to an enabled transition t in s: $EP(s)=\{t\}$,
2. Add to EP all the transitions which *could* produce sequences containing a transition which is dependent with respect to t,
3. Return all the transitions of EP enabled in s.

Transitions Selection Techniques. The sizes of the persistent-sets given by the previous algorithm heavily depends on the transitions selection technique. We have been interested in three selection techniques: the Godefroid and Wolper one [GW92], the Overman method [Ove81] and the algorithm of Valmari [Val91b] (the latter one computes stubborn-sets, but as we have explained above, their enabled-transitions subsets are also persistent-sets [God96b]). These three algorithms differ in the way the transitions are selected in the persistent-set EP:

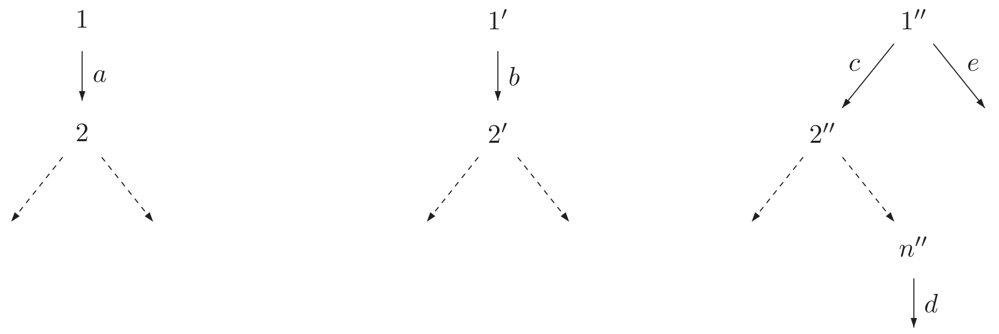
[GW92] For each transition t in EP, all the transitions which are enabled in s and dependent with respect to t are added to EP. If there exists a transition dependent with respect to t which is not enabled in s, then all the enabled transitions in s are added to EP.

[Ove81] For each transition t in EP, for each transition t' dependent with respect to t, all the enabled transitions of the transition system where t' appears are added to EP.

[Val91b] For each enabled transition t in s which is also in EP, all the transitions which are dependent with respect to t are added to EP. For each transition t not enabled in s, but which appears in EP, all the transitions which can make t enabled in a reachable state s' are added to EP.

The following example (from [Pag96]) makes these techniques more clear:

Example 1
Assume that one wants to explore the state-space of the following system composed of 3 transition systems, and that a and d are dependent, whereas all the others are independent.



We are just interested here in the persistent-sets computed for state $s = (1, 1', 1'')$. *Here are the results given by the three algorithms, when one first adds transition* a *to* $EP(s)$*:*

Technique	*[GW92]*	*[Ove81]*	*[Val91b]*
$EP(s)$	$\{a, b, c, e\}$	$\{a, c, e\}$	$\{a, d, c\}$
$EP(s) \cap enabled(s)$			$\{a, c\}$

The three previous techniques are ordered from the less to the most refined one. Of course, the more refined is the method, the greater is the cost of computation. Notice that using the smallest persistent-set is nothing more than an heuristic: it does not ensure to lead to the smallest state-space exploration.

Godefroid proved that [God96a]:

- No persistent-set computed by the algorithm of Overman is larger than all of those computed by the algorithm of Godefroid and Wolper,
- There exists a persistent-set (stubborn-set) given by the algorithm of Valmari which is smaller or equal to all of those computed by the way of the algorithm of Overman.

Selective State-Space Exploration with Persistent-Sets. A persistent-set of transitions is computed in each state reached during the state-space exploration. From each one of these states, we only execute the transitions which are in its persistent-set. Such an exploration is said to be a *selective state-space exploration*.

```
Stack.push(s_0); H := ∅
while Stack ≠ ∅ do
  s := Stack.pop()
  if s ∉ H then
    H := H ∪ {s}
    compute EP(s)
    for each t ∈ EP(s) ∩ enabled(s) do
      s' := succ_t(s)
      Stack.push(s')
    end for
  end if
end while
```

Algorithm 2: Selective state-space exploration using persistent-sets.

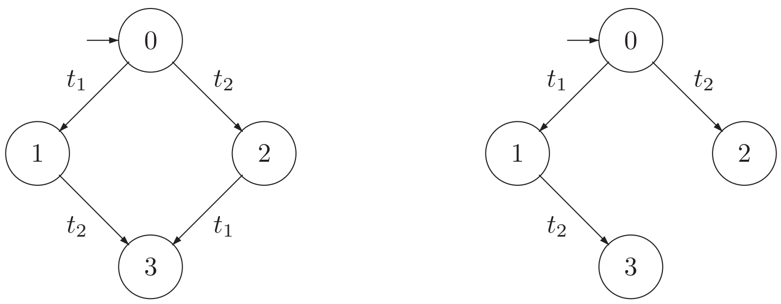
Algorithm 2 presents an algorithm taking advantage of this method where $EP(s)$ is the persistent-set computed in state s (it is adapted from [Pag96]).

About the Persistent-Sets Method. This method avoids one to explore the whole state-space of the transition system one want to model-check. Thus, we can expect saving memory if we only produce (or explore) a reduced automaton, and saving time because we explore less paths. One can find in [God96b] a complete description of the techniques described above and a comparison of the results and the complexity. It is also shown in [God96b] that every reachable deadlock state is preserved by the method. Many other properties can be checked by considering not only one but a subset of all the interleavings that preserve them (e.g., [HGP92, Pel93, GW93, Val93, WW97]).

3.2.2 Sleep-Sets Method

This second method is based on the selection of independent transitions and the fact that it is sometimes not necessary to explore some transitions because they lead to states which have already been visited during the exploration.

For example, consider the two automata of Figure 3. The first one (a) represents the state-space reached during a basic exploration, whereas the second one (b) shows a selective state-space exploration where one can avoid to fire a transition.



(a) Basic exploration.

(b) Selective exploration (sleep-sets technique).

Figure 3: Automata with two independent transitions.

Indeed, we assume that t_1 and t_2 are two independent transitions of an automaton, both enabled in a state 0, and we perform a depth-first exploration starting in 0. Let first execute t_1, leading from state 0 to state 1 and then t_2 finally leading to state 3. Then, we return to state 0 where we execute t_2. State 2 is now the current state and the transition t_1 is enabled there. But this transition leads from 2 to 3, which has already been visited. So, it is *sometimes* not necessary to do this twice and this is the aim of this method.

Sleep-Set. A *sleep-set* stores the transitions that lead to states which have already been visited [God91, Pag96]:

Definition 4 *A* sleep-set *for a state s is a set of transitions independent with respect to each other in s, which lead to states which have already been visited.*

Computation. A sleep-set is computed during the state-space exploration as it relies on the past of the current state-space exploration. Indeed, the sleep-set of a state s' depends on the sleep-set of the previous state s and on the transition t which leads from s to s' ($s \xrightarrow{t} s'$). Unlike persistent-sets, sleep-sets are computed from dynamic informations.

Here is a general algorithm for the computation of the sleep-set associated with a state s' reached by the execution of a transition: $s \xrightarrow{t} s'$ [Pag96].

1. clear $sleep(s')$.
2. Add to $sleep(s')$ all the transitions in $sleep(s)$ which are independent with respect to t.
3. Add t to $sleep(s)$ (memorization of the transitions which have already been executed from s).

Selective State-Space Exploration with Sleep-Sets. Algorithm 3 performs a "selective exploration" using sleep-sets in order to optimize it. Each time a new state is reached, one computes the associated sleep-set. All the enabled transitions but the asleep ones are executed from a new state.

```
Stack.push(s_0); H := ∅
while Stack ≠ ∅ do
  s := Stack.pop()
  if s ∉ H then
    H := H ∪ {s}
    for each t ∈ enabled(s) ∖ sleep(s) do
      s' := succ_t(s)
      compute sleep(s')
      Stack.push(s')
    end for
  end if
end while
```

Algorithm 3: Selective state-space exploration using sleep-sets.

About the Sleep-Sets Method. Notice that the sleep-sets technique does not allow one to reduce the number of visited states, but only the number of transitions which are explored. The result is that we cannot expect any reduction in memory space, but only in time space needed for the exploration.

Furthermore, this method is not suitable for reduced state-space generation, as is: for example, state 2 in Figure 3(b) becomes a deadlock state.

The sleep-sets method has been improved (e.g., *conflict-sets* [HGP92]) or adapted in several frameworks (e.g., [Pel93]).

Finally, these two techniques (persistent-sets and sleep-sets) can sometimes be used together in order to go further in the reductions. For example, Godefroid and Wolper have written an algorithm for deadlock states detection which uses both methods together [GW93].

4 Application of Partial-Order Methods to Electre

We now aim at building reduced FIFFO-lists. Therefore, we have been looking for known works dealing with partial-order methods and reduced state-space generation (e.g., [Val91b, Pel93]). As we will see later, such persistent-sets based techniques are not suitable in our framework, so we have also been looking for other partial-order methods or different uses of them (e.g., [God91, GW91, Val93, Pel94, GPS96]), one of which revealed to be more accurate.

4.1 Principle of the Reduction

Let us introduce our idea with an example. Consider the Electre program Foo, in Figure 1 (Page 6) whose associated FIFFO-list is pictured in Figure 4. Its transitions labeled $-e$ represent a dequeue of e, those which are labeled $+e$ represent an enqueue of e, and transitions labeled τ are only used for synchronization purpose (see Section 2.3).

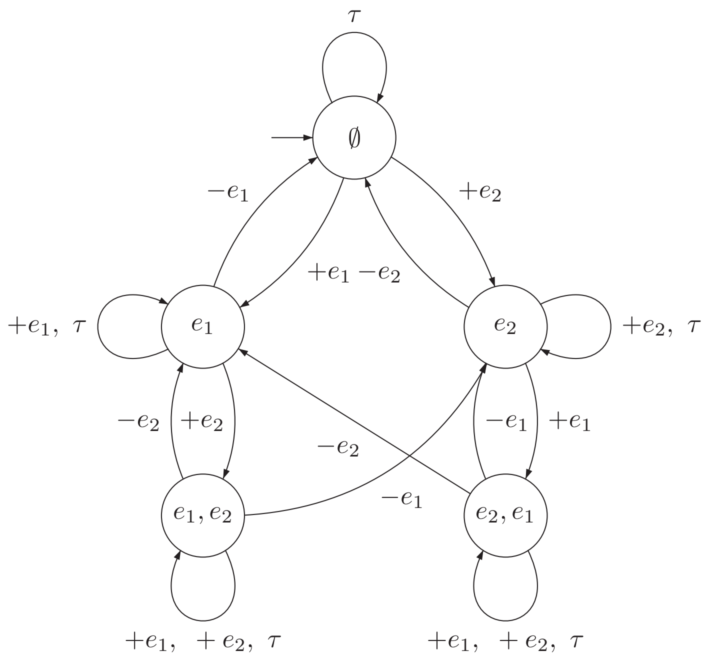


Figure 4: FIFFO-list for the events e_1 and e_2 (5 states, 21 transitions).

This program reacts to the instances of events e_1 and e_2, which can be memorized once. Thus, at any time, the memorizing device has one of the following contents:

$$(\emptyset) \quad (e_1) \quad (e_2) \quad (e_1, e_2) \quad (e_2, e_1)$$

As pictured in Figure 4, each possible content for the list corresponds to a state of its automaton. Intuitively, program Foo can take into account both events e_1 and e_2 immediately. And both sequences e_1e_2 and e_2e_1 lead to the same state from the given one, as shown in Figure 2 (Page 7). The order between e_1 and e_2 is therefore irrelevant, and these two events are independent with respect to each other (see Definition 1).

It clearly appears that the distinction between the two interleavings e_1, e_2 and e_2, e_1 is not necessary nor useful for the program execution. This gives a way to reduce the FIFFO-list: one replaces its two states (e_1, e_2) and (e_2, e_1) by a single state representing both of them. Since e_1 and e_2 are independent, we have: $[e_1e_2] = \{(e_1, e_2)\,, (e_2, e_1)\}$, then, it is possible to replace both states (e_1, e_2) and (e_2, e_1) by their trace $[e_1e_2]$.

Figure 5 shows the reduced automaton corresponding to the transition system in Figure 4. The suppressed parts are pictured in gray and dashed lines, whereas new parts are in bigger and bolder style. Notice that the state $[e_1e_2]$ in the new automaton stands for both states (e_1, e_2) and (e_2, e_1) in the previous one. By the way, one state and five transitions have been removed.

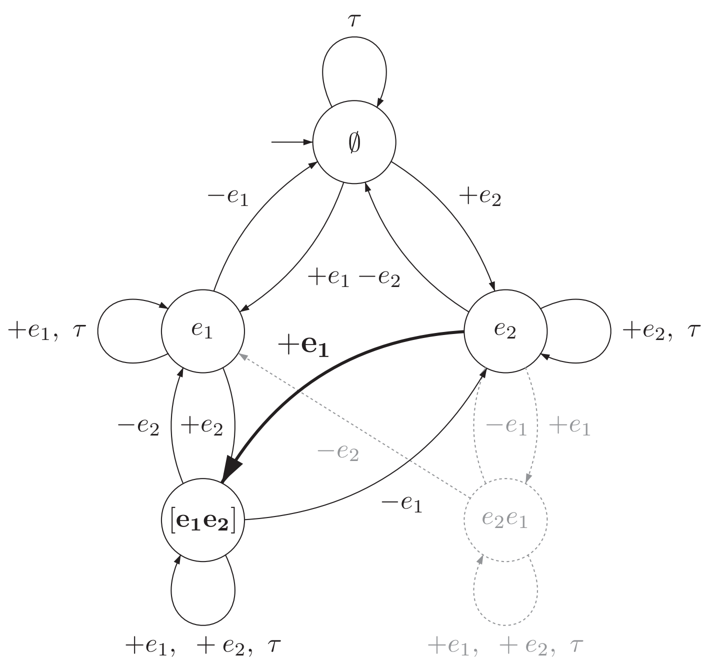


Figure 5: Reduced FIFFO-list for the independent events e_1 and e_2 (4 states, 16 transitions).

4.2 Computation of the Dependence Relation for Electre Programs

Our technique relies upon the fact that some events are independent with respect to each other, and others are not. We now need to be able to establish which events are dependent of which other ones. This can be done from a *dependence relation* (the definition is adapted from [GW93]).

Definition 5 *A* dependence relation $\Re$ *is a binary relation such as for any couple of events* e_1 *and* e_2*:*

- *either* $(e_1, e_2) \in \Re$*, then* e_1 *and* e_2 *are* dependent *(see Definition 1) with respect to each other,*
- *or* $(e_1, e_2) \notin \Re$*, then* e_1 *and* e_2 *are said to be* independent*.*

Notice that this is a symmetrical, but neither reflexive, nor transitive relation. Usually, $\Re$ is built reflexive, but here it is not necessary nor useful because each event appears at most once in any FIFFO-list.

A control automaton associated with an Electre program is computed by rewriting the program while taking into account the events it contains (see Section 2.2.2). Then, dependence between some events will imply that taking one of them into account will disable or enable the others, or that the sequences built on these events will not lead to the same states (see Definition 1). Thus, all the dependences between a program events can be computed on the control automaton. One only needs to explore this automaton and determine if Definition 1 holds every time several memorizable events can be taken into account. Meanwhile, this method has an important drawback: it needs a complete exploration of the control automaton.

We have developed another method which allows one to compute the dependence relation directly from the Electre program. Consequently, it is sufficient to analyze the structures of the program as explained in the following. The dependence relation is then obtained in a linear (in the number of Electre operators appearing in the program) time complexity.

Lemma 1 describes the cases of dependence and independence between events using the structures of the Electre language.

Lemma 1 *Let $< A >$, $< B >$ and $< C >$ be, respectively, a module structure and two event structures. We have for the following* Electre *operators (see Appendix A, Page 29):*

1. *"$< A >$ `await` $< B >$". All the events of $< A >$ are dependent with respect to* all the first level events[5] *of $< B >$,*

2. *"$< B >$ `or` $< C >$". All the events of $< B >$ are dependent with respect to* all the events *of $< C >$.*

Otherwise, they are independent.

Proof sketch.
This lemma can be proved by induction on the rewriting rules defined for each one of the Electre operators (see Appendix A and furthermore [CR95]).□

For any given Electre program, the dependence relation is computed according to Lemma 1. Each time the Electre program corresponds to one of the cases above, we add a couple of events (a *dependence*) per event of $< B >$ and per event of $< C >$ (resp. per event of $< A >$ and per first-level event of $< B >$) to the dependence relation. This is done recursively on $< B >$ and on $< C >$ (resp. $< A >$), since an Electre program is seen as its parse tree (see Section 2.2.1).

Notice that the example chosen in order to illustrate our results (see program Foo in Figure 1) does not contain dependent events. We would have preferred to show an example with both dependent and independent events. But such an Electre program must contain at least three events and then, leads to a non-reduced FIFFO-list with 16 states and 97 transitions. Obviously, this automaton cannot be pictured clearly.

4.3 Building of the Reduced FIFFO-List

We now describe our method for the construction of a reduced FIFFO-list. First, we choose a partial-order method well-suited for the desired reduction. Then, we exhibit our

[5]The *first-level events* associated with a given Electre program are all the events that can immediately be taken into account. For example, in the following Electre program, "`await` $\{e_1$ `launch` $A \,||\, e_2$ `launch` $[B$ `await` $e_3]\}$", e_1 and e_2 are first-level events, whereas e_3 is not.

algorithms.

4.3.1 Choice of a Partial-Order Method

The reduction must preserve the FIFFO-list behaviors. For example, the expected reduced FIFFO-list associated with the automaton depicted in Figure 4 (Page 17) is shown in Figure 5 (Page 18). We first examine the suitability of the persistent-sets method for that purpose, then the sleep-sets one.

Building a Reduced Automaton with a Persistent-Sets Method. First, consider the persistent-sets method on the automaton of Figure 4. The reduced automaton produced by the way of the persistent-sets technique is drawn in Figure 6(a) - where persistent-sets are written for each state between braces - together with the expected automaton in Figure 6(b).

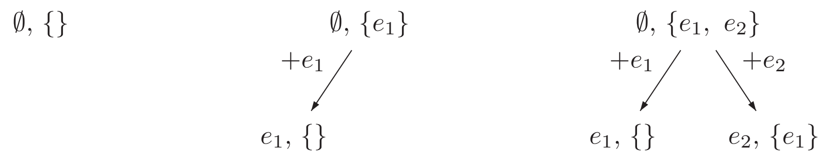
(a) Reduced FIFFO-list (persistent-sets technique).

(b) Expected reduced FIFFO-list.

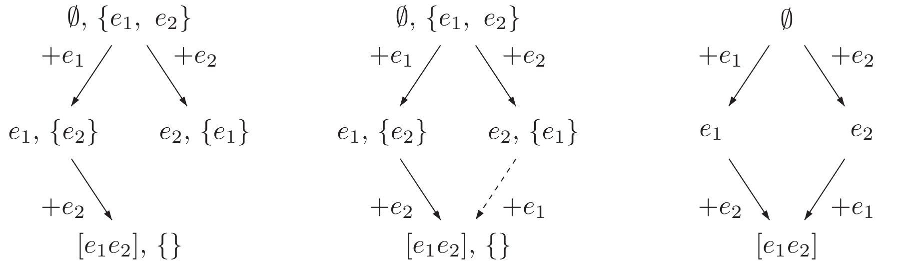
Figure 6: Produced and expected FIFFO-lists (skeletons).

Obviously, the achieved result is not what we were expecting: this reduced FIFFO-list does not allow the memorizing device to only contain an instance of event e_2. The persistent-sets method cannot be applied in order to get the reduction we are looking for. Indeed, we only select e_1 in the persistent-set of state 0 because e_1 and e_2 are independent. This means that one does not need to take care of e_2 in this state. But we could only have to memorize e_2. Our problem is due to the fact that the persistent-sets method preserves the traces but not the states, and as explained in Section 3, it omits some behaviors whereas we need all of them.

Producing a Reduced FIFFO-List with the Sleep-Sets Method. We now try to build, the reduced FIFFO-list corresponding to the automaton of Figure 4 by the way of the sleep-sets technique.



(a) Initially: the FIFFO-list is empty. (b) Enqueueing e_1. (c) Enqueueing e_2.



(d) Enqueueing e_2 (state e_1). (e) Enqueueing e_1 (state e_2): processing the sleep-set. (f) Reduced FIFFO-list (sleep-sets technique).

Figure 7: Building the reduced FIFFO-list with sleep-sets method (example).

Of course, the sleep-sets technique applies very well in our framework, and it gives the expected solution. Figure 7 shows the process to build the reduced FIFFO-list. It is just a kind of selective state-space exploration using the sleep-sets method (see Algorithm 3) but, here, we produce the reduced automaton during its exploration. The dequeue transitions are not shown on these automata because they are not necessary for the explanation, and the sleep-sets are written between braces for each state. We see in Figure 7(e) that the sleep-set of state e_2 contains an occurrence of event e_1. It means that there exists an already built state which is equivalent to the state we *would reach* by a memorization of event e_1. So, we only need to create a transition towards this existing state, avoiding, by the way, the creation of a new state, hence the reduction. This is the principle of our algorithm.

4.3.2 Algorithms for the Construction of the Reduced FIFFO-List

We assume that we have got a list of events ($list_{Evt}$), each of them can be stored at most once in the memorizing device. Algorithm 4 is the general algorithm for construction of the reduced automaton.

We use a queue in order to store the states that have to be treated later. It is named

```
queue_NextStates.enqueue (∅, {})
list_States.append(∅)
while queue_NextStates ≠ ∅ do
   state ← queue_NextStates.dequeue()
   list_NextStates ← state.computeNextStates()
   for each state_Next ∈ list_NextStates \ list_States do
      queue_NextStates.enqueue(state_Next)
      list_States.append(state_Next)
   end for
end while
```

Algorithm 4: Building the reduced FIFFO-list.

$queue_{NextStates}$. The states that have already been computed are stored in a list: $list_{States}$. Each one of the states is a structure containing the content of the memorizing device and a sleep-set: $state = (memorizing\ device\ content, sleep - set)$.

The main part of the algorithm is represented by the computation of the successors of a state. Algorithm 5 is devoted to this task.

For each state s:

1. First, we examine its sleep-set. For each asleep event, it consists in searching for a state equivalent to the one we would have reached by adding the event. And then, we establish a transition between them,

2. Then, we compute the successors of our state which are reached by adding the events which are neither in the sleep-set of s, nor in its memorizing device content,

3. Finally, we add the dequeueing transitions and the loops used for synchronization with the FIFFO-transition system (see Section 2.3).

```
/* Processing the sleep-set */
for each event e ∈ s.sleepSet do
  Search in list_States the state s' equivalent to s.memorizingDeviceContent ∪ {e}
  Connect s to s' through transition (s, +e, s')
end for

/* Computing the successors */
for each event e ∈ list_Evt \ (s.sleepSet ∪ s.memorizingDeviceContent) do
  s' ← inherit(s,e)
  Connect s to s' through transition (s, +e, s')
end for

/* Adding loops for events that have already been stored */
for each event e ∈ s.memorizingDeviceContent do
  Connect s to s through loop (s, +e, s)
end for

/* Adding the τ loop */
Connect s to s through loop (s, τ, s)

/* Add the dequeue transitions */
for each event e ∈ s.memorizingDeviceContent do
  Search in list_States a state s' equivalent to s.memorizingDeviceContent \ {e}
  Connect s to s' through transition (s, −e, s')
end for

Return the list of all the created successors states
```

Algorithm 5: Function computeNextStates: computation of the successors of a state $s = (\mathit{memorizing\ device\ content}, \mathit{sleep\ set})$.

Now, we only need to describe the function "inherit" which computes the successor s' of a state s, reached by a transition labeled by event e. This function first computes the memorizing device content associated with s'. It only adds e to the content of s: $s'.\mathit{memorizingDeviceContent} \leftarrow s.\mathit{memorizingDeviceContent} \cup \{e\}$. Then, it computes the sleep-set of s' according to the algorithm given in Section 3.2.2.

4.4 Complexity Issue and Results

The structure of the *unreduced* FIFFO-list is specific. Each one of its states corresponds to a *permutation* of r events among n ($r \leq n$). As a consequence, the complexity of the algorithm could seem very easy to determine, but as we will see, it is not actually the fact. Our algorithm has been implemented in Objective CAML and the resulting tool has been used during the verification phase of an embedded software dedicated to the control program of an aircraft engine. The results are given in Section 4.4.2.

4.4.1 Complexity Issue

The most interesting characteristic of the algorithms that produce reduced transition systems is the amount of reduction they provide. Hence, we aim at expressing the complexity of our method as the number of states of the generated reduced automaton.

Theoretical Approach of Complexity. As explained in Section 4.1, each state of the unreduced automaton is a r ordered outcome from a permutation of n events ($r \leq n$) and each permutation of $r \in [1, n]$ events among n is a state of the unreduced automaton. So the number of states of the unreduced automaton is:

$$\mathcal{N}_{ur} = \sum_{r=0}^{n} r! \binom{n}{r} = \sum_{r=0}^{n} \frac{n!}{(n-r)!}$$

Moreover, if all the program events are dependent, no reduction can be expected. In this case, in fact the *worst* case, the "reduced" automaton and the unreduced one are identical. Thus, the number of states in the worst case is:

$$\mathcal{N}_{worst} = \mathcal{N}_{ur} = \sum_{r=0}^{n} r! \binom{n}{r}$$

At the opposite, when all the events are independent, the reduction is maximal. In this case, the *best* one, there is no matter in the event order. Thus, the complexity falls from a permutation down to a combination. Therefore, the number of states of the best-reduced automaton is given by:

$$\mathcal{N}_{best} = \sum_{r=0}^{n} \binom{n}{r} = \sum_{r=0}^{n} \frac{n!}{r! \times (n-r)!}$$

The last case is the *average* one. Despite the combinatorial structure of the automaton and the fact that both previous cases were easy, we were not able to find an expression for the average complexity. First, notice that unlike the best and the worst cases, all the events do not behave in the same way. Each event can now only permute with the events of which it is independent. One can imagine modelling this by a number of dependent events and a number of independent ones linked to each event. But this is not sufficient because the most important difference between the best case and the worst case on the one hand, and the average case on the other hand, is the difficulty to model the order the events appear in. Example 2 shows its big importance.

Example 2
We consider the three events e_1, e_2, e_3 linked by the following dependence relation: $\Re = \{(e_1, e_2), (e_1, e_3)\}$ obtained from an Electre *program. The two following permutations of these three events show that, the number of their equivalent sequences of events depend on the order:*

- (e_1, e_2, e_3) *has an equivalent permutation:* (e_1, e_3, e_2),
- *rather than,* (e_2, e_1, e_3) *does not have any.*

Overview of the Complexity on Some Examples. Since we did not manage to find an expression for the average case complexity, we now give a flavour of it. Two parameters step in our method: the number of events (all independent) and the percentage of dependence between events. So, we look at two test phases, one for each of these parameters, and we make them vary.

As our evaluation criterion is the number of suppressed states, we give in Tables 1 and 2 (one per test phase) the number of states of the rough automaton, the number of states of the reduced one, and the percentage of suppressed states gathered with the Electre programs of these tests. Notice that these programs are only examples, and a lot of others, having the same characteristics (number of events and percentage of dependent events), could have been chosen and would have produced the same results.

First Test. Consider 5 Electre programs with an increasing number of independent events. Table 1 shows both test programs and the achieved results.

Notice that the percentage of suppressed states increases quickly. This can be explained in the following way: consider a program containing the five independent events: e_1, e_2, e_3, e_4 and e_5. One can build 120 (5!) different ordered sequences containing each of these 5 events once. But all these sequences have the same trace: $[e_1e_2e_3e_4e_5]$, due to the empty dependence relation. Thus, in the reduced automaton, this unique last state stands for the 120 others in the rough automaton.

# events	Program	# states (rough)	# states (reduced)	% reduction
1	`await` $e1$	2	2	0%
2	`await` $\{e1 \,\|\|\, e2\}$	5	4	20%
3	`await` $\{e1 \,\|\|\, e2 \,\|\|\, e3\}$	16	8	50%
4	`await` $\{e1 \,\|\|\, e2 \,\|\|\, e3 \,\|\|\, e4\}$	65	16	75%
5	`await` $\{e1 \,\|\|\, e2 \,\|\|\, e3 \,\|\|\, e4 \,\|\|\, e5\}$	326	32	90%

Table 1: Percentage of suppressed states with respect to the number of events.

Second Test. Now, in order to evaluate the influence of the percentage of event dependences, consider 5 programs written in the Electre language, all of them containing the same 5 events, but having an increasing percentage of dependence: from 0% to 100%. This simply comes from:

- First, a program where all the events are composed in a parallel preemption structure, making them independent,
- Then, the parallelism operators ("||") are gradually replaced by exclusive operators ("or"). This increases the percentage of dependent events (see Lemma 1).

Table 2 shows the programs written to get the variation of the percentage of dependence together with the resulting numbers.

% dependent events	Program	# states (rough)	# states (reduced)	% reduction
0%	$\texttt{await}\,\{e1 \,\|\|\, e2 \,\|\|\, e3 \,\|\|\, e4 \,\|\|\, e5\}$	326	32	90%
20% $(1-\frac{4}{5})$	$\texttt{await}\,\{e1\,\texttt{or}\,\{e2 \,\|\|\, e3 \,\|\|\, e4 \,\|\|\, e5\}\}$	326	40	88%
40% $(1-\frac{3}{5})$	$\texttt{await}\,\{e1\,\texttt{or}\,e2\,\texttt{or}\,\{e3 \,\|\|\, e4 \,\|\|\, e5\}\}$	326	64	80%
60% $(1-\frac{2}{5})$	$\texttt{await}\,\{e1\,\texttt{or}\,e2\,\texttt{or}\,e3\,\texttt{or}\,\{e4 \,\|\|\, e5\}\}$	326	130	60%
100%	$\texttt{await}\,\{e1\,\texttt{or}\,e2\,\texttt{or}\,e3\,\texttt{or}\,e4\,\texttt{or}\,e5\}$	326	326	0%

Table 2: Percentage of suppressed states with respect to the percentage of dependent events.

If all the events are dependent with respect to each other, or if there is one or no event appearing in an Electre program, then any reduction can be expected. Indeed, in these cases the computed traces contain one and exactly one sequence of events (the number of traces is equal to the number of sequences).

4.4.2 Results

Consider the program Foo in Figure 1 (Page 6) and the FIFFO-lists associated with it: the rough one (in Figure 4, Page 17) and the reduced one (see Figure 5, Page 18). Here, our reduction technique allows removing 1 state (20% of the total number of states) and 5 transitions.

A Real-Case Study. Nevertheless, program Foo is an academic example, and moreover, a very small but readable one. The tool implementing our method has been used during the verification phase of an embedded software dedicated to the control program of an aircraft engine.

This case study is more deeply described in [Boi99] from which the following main features and results were extracted. It is mainly composed of two tasks: the first one is dedicated to input/output control (`IO`), and the second one to computation (`CMP`). Both of them are divided in several programs: `IO1, IO2, IO3` for the input/output control task and `CMP1, CMP2` for the computer. Furthermore, their communications are handled by two tasks: `IO_CMP` and `CMP_IO`. Table 3 shows for each program: its number of operators, its number of events, the percentage of dependence between its events, and the number of states of its non-reduced associated FIFFO-list.

The dependence relation has been computed for each one of these programs using the method given by Lemma 1. The time complexity of this method is linear in the number of Electre operators appearing in the program. Thus, despite the high number of operators appearing in these programs, the time needed for the computation of the dependence relation is negligible[6].

[6]It is not comparable to the time needed to compute the relation using an exploration of the FIFFO-transition system. For example, the control transition systems associated with programs `IO1` and `CMP1`

Program	# Electre operators	# Events	Percentage of dependence	Rough FIFFO-list # states
IO1	60	10	3.33%	9,864,101
IO2	12	11	100%	108,505,112
IO3	10	9	100%	986,410
CMP1	49	15	2.86%	$\simeq$ 3,550,000,000,000
CMP2	14	13	100%	16,926,797,486
IO_CMP	14	13	100%	16,926,797,486
CMP_IO	12	11	100%	108,505,112

Table 3: Overview of the software.

Unfortunately, most of the programs have strongly dependent events, particularly, one cannot expect any reduction of the FIFFO-lists associated to the programs `IO2, IO3, CMP2, IO_CMP` and `CMP_IO`, since all their events are dependent.

Program	Reduced FIFFO-lists # states	Percentage of suppressed states
IO1	2,048	99,98%
CMP1	133,120	99,99%

Table 4: Reduced FIFFO-lists.

But, both programs, `IO1` and `CMP1`, have a very low amount of dependent events (respectively 3.33% and 2.86%). So, our method has been successfully applied on these programs, and the rate of suppressed states between the unreduced automata on the one hand, and the reduced one on the other hand is amazing. We figure out the results in Table 4.

5 Conclusion

We have shown in this paper a model of reactive applications with event memorization. This is a meaningful and important feature of the asynchronous reactive language Electre. The memorizing device is itself modelled by a FIFFO-list which is bounded, since at most one occurrence of each event is stored. But it is usually far too large. We focused on some verification issues on this model and more accurately on the reduction of the event memorization space.

Our Contribution. Due to the special structure of a FIFFO-list (each one of its state is equivalent to the interleaving which leads from the initial state to this one), we have been

have got 233 states/3,359 transitions and 9,651 states/212,766 transitions respectively.

able to produce reduced FIFFO-lists by the means of a partial-order method. Since these techniques rely upon the equivalence of interleavings of actions, and lead to the suppression of all but one equivalent interleavings, it is quite an amazing result in our case. Indeed, we were not able to remove any state or behavior from the FIFFO-list.

Our technique uses concurrency in Electre programs as a starting point and idea for a reduction, and particularly sleep-sets method for the detection and merging of equivalent states.

Of course, the reduction does not preserve all the properties of the FIFFO-list. For example, the order between the stored events is lost, and properties such as "Is every occurrence of e_1 always preceded by an occurrence of e_2" can not be safely checked on the reduced model. But, this reduction preserves all the behaviors of the FIFFO-list. Furthermore, it leads to drastic reductions in the number of states as shown in the real-case study.

Future Work and Perspectives. The next step of our researches in this area lies in the proof of bisimulation between the FIFFO-transition system synchronized with the rough FIFFO-list on the one hand, and the FIFFO-transition system synchronized with the reduced FIFFO-list on the other hand. Such a result would mean that our reduction does not only preserves all the behaviors of the system, but it does not create any wrong ones.

Our technique could also be improved in order to make it possible to check some properties dealing with the order between the stored events occurrences. For example, the following property: "Is every occurrence of e_1 always preceded by an occurrence of e_2", could be checked in the reduced model, if the order between the occurrences of e_1 and e_2 had been kept. We only need to specify *independently* from the Electre program, that e_1 and e_2 are dependent events.

Finally, one appealing question is to study the unbounded FIFFO-lists. Indeed, it is possible to endow the events with the property of *unbounded memorization*. It would then be challenging to try to combine the partial-order techniques with abstraction techniques, in order to deal with such FIFFO-lists.

A SOS Rules for the Semantics of Electre

We give hereafter the SOS rules which give the semantics of the Electre language. For the commutative operators $\|, \mathsf{or}$, we give only half of the rules: the versions obtained by commuting the variables hold as well.

1. modules and events identifiers:

   (a) $\dfrac{end_m \in \mathcal{E}}{\mathcal{E} \vdash m : \mathsf{Mid} \rightarrow nil : \mathsf{NIL}}$

   (b) $\dfrac{end_m \notin \mathcal{E}}{\mathcal{E} \vdash m : \mathsf{Mid} \rightarrow m : \mathsf{Mid}}$

   (c) $\dfrac{e \in \mathcal{E}}{\mathcal{E} \vdash (@)e : \mathsf{Eid} \rightarrow nil : \mathsf{NIL}}$

   (d) $\dfrac{e \notin \mathcal{E}}{\mathcal{E} \vdash (@)e : \mathsf{Eid} \rightarrow (@)e : \mathsf{Eid}}$

2. `or` operator:

   (a) $\dfrac{\mathcal{E} \vdash E_1 : \mathsf{ES} \rightarrow E_1' : \mathsf{ES} \quad E_2 : \mathsf{ES} \rightarrow E_2' : \mathsf{ES}}{\mathcal{E} \vdash E_1 \, \mathtt{or} \, E_2 : \mathsf{ES} \rightarrow E_1' \, \mathtt{or} \, E_2' : \mathsf{ES}}$

   (b) $\dfrac{\mathcal{E} \vdash E_1 : \mathsf{ES} \rightarrow X_1' : \theta_1' \quad E_2 : \mathsf{ES} \rightarrow E_2' : \mathsf{ES} \quad \theta_1' \in \{\mathsf{MS}, \mathsf{NIL}\}}{\mathcal{E} \vdash E_1 \, \mathtt{or} \, E_2 : \mathsf{ES} \rightarrow X_1' : \theta_1'}$

3. $||$ operator for *events*:

   (a) $\dfrac{\mathcal{E} \vdash E_1 : \mathsf{ES} \rightarrow E_1' : \mathsf{ES} \quad E_2 : \mathsf{ES} \rightarrow E_2' : \mathsf{ES}}{\mathcal{E} \vdash E_1 \, || \, E_2 : \mathsf{ES} \rightarrow E_1' \, || \, E_2' : \mathsf{ES}}$

   (b) $\dfrac{\mathcal{E} \vdash E_1 : \mathsf{ES} \rightarrow P_1' : \mathsf{MS} \quad E_2 : \mathsf{ES} \rightarrow E_2' : \mathsf{ES}}{\mathcal{E} \vdash E_1 \, || \, E_2 : \mathsf{ES} \rightarrow P_1' \, || \, \mathtt{await} \, E_2' : \mathsf{MS}}$

   (c) Symmetrical rule.

   (d) $\dfrac{\mathcal{E} \vdash E_1 : \mathsf{ES} \rightarrow nil : \mathsf{NIL} \quad E_2 : \mathsf{ES} \rightarrow E_2' : \mathsf{ES}}{\mathcal{E} \vdash E_1 \, || \, E_2 : \mathsf{ES} \rightarrow E_2' : \mathsf{ES}}$

4. $||$ operator for *modules*:

   (a) $\dfrac{\mathcal{E} \vdash P_1 : \mathsf{MS} \rightarrow P_1' : \mathsf{MS} \quad P_2 : \mathsf{MS} \rightarrow P_2' : \mathsf{MS}}{\mathcal{E} \vdash P_1 \, || \, P_2 : \mathsf{MS} \rightarrow P_1' \, || \, P_2' : \mathsf{MS}}$

   (b) $\dfrac{\mathcal{E} \vdash P_1 : \mathsf{MS} \rightarrow nil : \mathsf{NIL} \quad P_2 : \mathsf{MS} \rightarrow P_2' : \mathsf{MS}}{\mathcal{E} \vdash P_1 \, || \, P_2 : \mathsf{MS} \rightarrow P_2' : \mathsf{MS}}$

5. `await` operator:

(a) $$\frac{\mathcal{E} \vdash P : \mathsf{MS} \rightarrow X' : \theta' \quad E : \mathsf{ES} \rightarrow E' : \mathsf{ES} \quad \theta' \in \{\mathsf{MS}, \mathsf{NIL}\}}{\mathcal{E} \vdash P\,\mathtt{await}\,E : \mathsf{MS} \rightarrow X'\,\mathtt{await}\,E' : \mathsf{MS}}$$

(b) $$\frac{\mathcal{E} \vdash P : \mathsf{MS} \rightarrow P' : \mathsf{MS} \quad E : \mathsf{ES} \rightarrow X' : \theta' \quad \theta' \in \{\mathsf{MS}, \mathsf{NIL}\}}{\mathcal{E} \vdash P\,\mathtt{await}\,E : \mathsf{MS} \rightarrow X' : \theta'}$$

6. `launch` operator:

(a) $$\frac{\mathcal{E} \vdash E : \mathsf{ES} \rightarrow E' : \mathsf{ES}}{\mathcal{E} \vdash E\,\mathtt{launch}\,P : \mathsf{ES} \rightarrow E'\,\mathtt{launch}\,P : \mathsf{ES}}$$

(b) $$\frac{\mathcal{E} \vdash E : \mathsf{ES} \rightarrow nil : \mathsf{NIL}}{\mathcal{E} \vdash E\,\mathtt{launch}\,P : \mathsf{ES} \rightarrow E'\,\mathtt{launch}\,P : \mathsf{ES}}$$

7. `sequence` operator:

(a) $$\frac{\mathcal{E} \vdash P_1 : \mathsf{MS} \rightarrow P_1' : \mathsf{MS} \quad P_2' : \mathsf{MS} \rightarrow P_2' : \theta}{\mathcal{E} \vdash P_1; P_2 : \mathsf{MS} \rightarrow P_1'; P_2 : \mathsf{MS}}$$

8. `loop` operator:

(a) $$\frac{\mathcal{E} \vdash P : \mathsf{MS} \rightarrow P' : \mathsf{MS}}{\mathcal{E} \vdash \mathtt{loop}\,P\,\mathtt{endloop} : \mathsf{MS} \rightarrow P'; \mathtt{loop}\,P\,\mathtt{endloop} : \mathsf{MS}}$$

References

[AFdR80] K. R. Apt, N. Francez, and W. P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, July 1980.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, september 1991.

[BBRR97] F. Boniol, A. Burgueño, O. Roux, and V. Rusu. Analysis of slope-parametric hybrid automata. *Lecture Notes in Computer Science*, 1201:75–??, 1997.

[Bd91] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, september 1991.

[Boi99] P. Boisieau. *Vérification et exécution d'applications temps-réel industrielles avec* ELECTRE. PhD thesis, Ecole Centrale de Nantes, 1999.

[BR99] P. Boisieau and O. Roux. Splitting reachability analysis of hybrid automata. In *Proc. 11th Euromicro Conference On Real-Time Systems*, pages 98–105, York, England, June 1999.

[CR95] F. Cassez and O. Roux. Compilation of the ELECTRE reactive language into finite transition systems. *Theoretical Computer Science*, 146(1–2):109–143, July 1995.

[EF82] T. E. Elrad and N. Francez. Decomposition of distributed programs into communicationclosed layers. *Science of Computer Programming*, 2(3), 1982.

[Esp94] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2–3):151–195, December 1994.

[GH85] M. G. Gouda and J. Y. Han. Protocol validation by fair progress state exploration. *Computer Networks and ISDN systems*, pages 353–361, May 1985.

[GHP95] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. *Formal Methods in System Design*, pages 1–15, November 1995. also in: Proc. CAV92, Montreal, Canada.

[GKPP99] R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial order approach to branching time logic model checking. *INFCTRL: Information and Computation (formerly Information and Control)*, 150, 1999.

[GL94] O. Grumberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[God91] P. Godefroid. Using partial orders to improve automatic verification methods. In E. M. Clarke, editor, *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV '90), Rutgers, New Jersey, 1990*, number 531 in Lecture Notes in Computer Science, pages 176–185, Berlin-Heidelberg-New York, 1991. Springer.

[God96a] P. Godefroid. On the costs and benefits of using partial-order methods for the verification of concurrent systems. In *Proceedings of DIMACS Workshop on Partial-Order Methods in Verification, AMS*, Princeton, 1996.

[God96b] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996.

[GP93] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In *Proceedings of the 5th International Conference on Computer Aided Verification, Greece*, number 697 in Lecture Notes in Computer Science, pages 409–423, Berlin-Heidelberg-New York, 1993. Springer.

[GPS96] P. Godefroid, D. Peled, and M. Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. *IEEE Transactions on software engineering*, 22(7), July 1996.

[GW91] P. Godefroid and P. Wolper. A partial approach to model checking. In *6th symposium on logic in computer science*, Amsterdam, 1991.

[GW92] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In Kim G. Larsen and Arne Skou, editors, *Proceedings of Computer Aided Verification (CAV '91)*, volume 575 of *LNCS*, pages 332–342, Berlin, Germany, July 1992. Springer.

[GW93] P. Godefroid and P. Wolper. Partial-order methods for temporal verification. *CONCUR '93 Proceedings Lecture Notes in Computer Science*, 715:233–246, August 1993.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow language LUSTRE. *Proceedings of the IEEE*, 79(9):1304–1320, september 1991.

[HGP92] G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. 12th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, Orlando, Fl., June 1992.

[Hol87] G. J. Holzmann. Automated Protocol Validation in *Argos*: Assertion Proving and Scatter Searching. *IEEE Transactions on Software Engineering*, 13(6):683–696, June 1987.

[HP94] G. J. Holzmann and D. Peled. An improvement in formal verification. October 1994.

[JZ93] W. Janssen and J. Zwiers. Specifying and proving communication closedness in protocols. In *Proc. 13th IFIP WG 6.1 International Symposium on Protocol Specification, Testing and Verification*, pages 323–339, Liège, May 1993. North-Holland.

[KP86] Y. Kornatzky and S. S. Pinter. A model checker for partial order temporal logic. Technical Report EE PUB 597, Departement of Electrical Enginering, Technion-Israel Institute of Technology, 1986.

[KP87] S. Katz and D. Peled. Interleaving set temporal logic. In *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 178–190, August 1987. See also Technical Report #505, Technion – Israel Institute of Technology, Department of Computer Science, Haifa, Israel, March 1988.

[KP92a] S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101(2):337–359, July 1992.

[KP92b] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6:107–120, 1992.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LBBG86] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. SIGNAL: a data-flow oriented language for signal processing. *IEEE transactions on ASSP*, ASSP-34(2):362–374, 1986.

[Maz86] A. Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II: Proceedings of an Advanced Course*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324, 1986.

[McM92] K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In G. v. Bochman and D. K. Probst, editors, *Proc. International Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer-Verlag, 1992.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MP93] Z. Manna and A. Pnueli. Models for reactivity. *Acta Informatica*, 30:609–678, 1993.

[Ove81] W.T. Overman. *Verification of concurrent systems: function and timing*. PhD thesis, Universisty of California, Los Angeles, 1981.

[Pag96] F. Pagani. Partial orders and verification of real-time systems. *Lecture Notes in Computer Science*, 1135:327–??, 1996.

[Pel93] D. Peled. All from one, one from all: on model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification, Greece*, number 697 in Lecture Notes in Computer Science, pages 409–423, Berlin-Heidelberg-New York, 1993. Springer.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Lecture Notes in Computer Science*, 818:377–??, 1994.

[Pen88] W. Penczek. A temporal logic for event structures. *Fundamenta Informaticae*, 11(3):297–326, 1988.

[Pen90] W. Penczek. Proving partial order properties using cctl. In *Proc. Concurrency and Compositionality Workshop*, San Miniato, Italy, 1990.

[PH85] A. Pnueli and D. Harel. *On the Development of Reactive Systems*, volume F 13 of *NATO ASI*, pages 477–498. Springer-Verlag Berlin Heidelberg, K.R. Apt edition, 1985.

[Pnu86a] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In W.-P. de Roever and G. Rozenberg, editors, *Current trends in Concurrency: Overviews and Tutorials*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, New York, N.Y., 1986.

[Pnu86b] A. Pnueli. Specification and development of reactive systems. In *Information Processing*. Elsevier Science Publishers B.V. (North Holland), 1986.

[Pra86] V. R. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, February 1986.

[PRH92] J. Perraud, O. Roux, and M. Huou. Operational semantics of a kernel of the language ELECTRE. *Theoretical Computer Science*, 97(1):83–104, april 1992.

[PW84] S. S. Pinter and P. Wolper. A temporal logic to reason about partially ordered computations. In *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, pages 28–37, Vancouver, August 1984.

[SdR89] F. A. Stomp and W. P. de Roever. Designing distributed algorithms by means of formal sequentially phased reasoning (extended abstract). In Jean-Claude Bermond and Michel Raynal, editors, *Distributed Algorithms, 3rd International Workshop*, volume 392 of *Lecture Notes in Computer Science*, pages 242–253, Nice, France, 26–28 September 1989. Springer.

[SFRC99] G. Sutre, A. Finkel, O. Roux, and F. Cassez. Effective recognizability and model checking of reactive fiffo automata. *Lecture Notes in Computer Science*, 1548:106–123, 1999.

[Val88a] A. Valmari. Error detection by reduced reachability graph generation. In *Proc. 9th International Conference on Application and Theory of Petri Nets*, pages 95–112, Venice, 1988.

[Val88b] A. Valmari. Heuristics for lazy state generation speeds up analysis of concurrent systems. In *Proc. of the Finnish Artificial Intelligence Symposium STeP-88*, volume 2, pages 640–650, Helsinki, 1988.

[Val91a] A. Valmari. A stubborn attack on state explosion. *Lecture Notes in Computer Science*, 531:156–??, 1991.

[Val91b] A. Valmari. Stubborn sets for reduced state space generation. *LNCS 483 : Advances in Petri Nets'90*, 1991.

[Val93] A. Valmari. On-the-fly verification with stubborn sets. In *Proc. 5th Conference on Computer Aided Verification*, volume 483 of *Lecture Notes in Computer Science*, pages 397–408. Springer-Verlag, Elounda, June 1993.

[Wes86] C. H. West. Protocol validation by random state exploration. In *Proc. 6th IFIP WG 6.1 International Symposium on Protocol Specification, Testing and Verification*, pages 233–242. North-Holland, 1986.

[Win86] G. Winskel. Event structures. In W. Brauer, editor, *Petri nets: central models and their properties; advances in Petri nets; proceedings of an advanced course, Bad Honnef, 8.-19. Sept. 1986, Vol. 2*, number 255 in Lecture Notes in Computer Science, Berlin-Heidelberg-New York, 1986. Springer.

[WW97] B. Willems and P. Wolper. Partial-order methods for model checking: From linear time to branching time. In *Actes de JBOPAD97*, June 1997.