Using Partial Orders to Improve Automatic Verification Methods (Extended Abstract)*

Patrice Godefroid
Université de Liège, Institut Montefiore, B28
4000 Liège Sart-Tilman, Belgium
Email: godefroid@montefiore.ulg.ac.be

Abstract

In this paper, we present a verification method for concurrent finite-state systems that attempts to avoid the part of the combinatorial explosion due to the modeling of concurrency by interleavings. The behavior of a system is described in terms of partial orders (more precisely in terms of Mazurkiewicz's traces) rather than in terms of interleavings. We introduce the notion of "trace automaton" which generates only one linearization per partial order. Then we show how to use trace automata to prove program correctness.

1 Introduction

Finite-state methods are quite widely used for concurrent program verification. Indeed, they have several advantages: they are simple and easy to understand and they can be fully automated. Unfortunately, these methods also have some serious drawbacks. They are not always applicable and, when they are applicable, they are often limited by combinatorial explosion.

The frustrating fact is that a lot of this combinatorial explosion is unnecessary: it is due to the modeling of concurrency by interleavings. For example, the concurrent composition of two n-state processes having completely independent activities is represented by an n^2-state process.

Of course, it has been recognized for some time that concurrency and nondeterminism are not the same thing. This observation has inspired a fairly large body of work on "partial order" models of concurrency [Lam78] [Maz86] [Pra86] [Win86]. With very few exceptions, work in this area is limited to rather abstract semantical models.

In this paper, we take a very pragmatic point of view towards partial order models. Our goal is to develop verification methods for concurrent finite-state systems that avoid the part of the combinatorial explosion due to the modeling of concurrency by interleaving. We present a framework in which this can be done successfully.

To define a verification method, four elements are necessary: a representation of programs, a representation of properties, a semantics according to which we compare programs and properties, and an algorithm for doing this comparison [Wol89].

For representing programs and properties, we chose one-safe place/transition-nets (P/T-nets) [Rei85] [Roz86]. This is a well-known formalism and it fits very well both with the interleaving and the partial order approaches. As semantic model, we use Mazurkiewicz's traces [Maz86]. However, we use this model in such a way that the results of our verification are identical to what one would obtain with an interleaving model of concurrency. Precisely, we verify that the language of firing sequences of the

*This research is supported by the European Community ESPRIT BRA project SPEC (3096).

one-safe P/T-net N_I representing the implementation is included in the language of firing sequences of the net N_S representing the specification.

Our verification method works in several steps. First, we build an automaton from the net representing the implementation. However, this automaton does not represent all interleavings. It only represents one interleaving for each Mazurkiewicz trace in the semantics of the implementation. This automaton can be very much smaller than the one representing all interleavings. We then compare this automaton to the net for the specification taking into account the dependency relation of the Mazurkiewicz trace semantics.

2 One-safe P/T-nets

Definition 2.1 *A one-safe place/transition-net (P/T-net), is a quadruple $N = (S, T, F, M_0)$ where S and T are finite, disjoint, nonempty sets of respectively* places *(*local states*) and* transitions*; $F \subseteq S \times T \cup T \times S$ is a* flow relation *such that $dom(F) \cup cod(F) = S \cup T$ (no isolated elements); and $M_0 \subseteq S$ is the* initial marking.

P/T-nets are represented graphically using boxes (or straight lines) to represent transitions, circles to represent places, and arrows to represent the flow relation. In such a representation, circles corresponding to places in the initial marking are marked with dots (tokens).

For each $x \in S \cup T$, the sets ${}^\cdot x = \{y : (y, x) \in F\}$, $x^\cdot = \{y : (x, y) \in F\}$, and ${}^\cdot x^\cdot = {}^\cdot x \cup x^\cdot$ are called respectively the *preset*, the *postset* and the *proximity* of x.

A *marking* M (or *global state*) of such a net N is a subset of S.

Definition 2.2 *Let M be a marking of N. A transition $t \in T$ is* M-firable[1] *iff $({}^\cdot t \subseteq M) \wedge ({}^\cdot t \cap (M \setminus {}^\cdot t) = \emptyset)$.*

Thus each place of each marking will contain at most one token. An M-firable transition $t \in T$ may fire and yield a *successor marking* M' of M which is such that $M' = (M \setminus {}^\cdot t) \cup t^\cdot$.

If t fires from M to M' we write $M\ [\ t > M'$. A *reachable marking* of N is a marking M such that $\exists t_1, \ldots, t_n \in T$: $M_0\ [\ t_1 > M_1\ [\ t_2 > M_2\ [\ \ldots\ [\ t_n > M_n = M$. The set of all reachable markings of N will be denoted by $mark(N)$.

The sequence $t_1, t_2, \ldots, t_n$ is called a *firing sequence* of N. The set of all firing sequences of N will be called the *firing sequence language of N*. This language is prefix closed.

Definition 2.3 *A one-safe P/T-net N is called* contact-free *iff for all $M \in mark(N)$ and for all $t \in T$: $({}^\cdot t \subseteq M) \Rightarrow ({}^\cdot t \cap (M \setminus {}^\cdot t) = \emptyset)$.*

3 Automata

A finite-state deterministic automaton is a quadruple (S, Σ, δ, s_0) where S is a finite set of states; Σ is an alphabet; $\delta : S \times \Sigma \rightarrow S$ is a deterministic transition function; and s_0 is the starting state.

The language generated by such an automaton is the set of words $w = a_1 a_2 \ldots a_n$ such that there exist $s_i = \delta(s_{i-1}, a_i)$, for all $i : 1 \leq i \leq n$. An automaton can be represented by a directed graph. The nodes of this graph represent the states of S while the edges represent the transition function and are labeled with elements of Σ.

It is easy to define an automaton that generates the firing sequence language of a given one-safe P/T-net N: $S = mark(N)$; $\Sigma = T$, the set of transitions of N; $\delta(M, t) = M'$ iff $M, M' \in mark(N)$ and $M\ [\ t > M'$; and $s_0 = M_0$, the initial marking of N. Clearly, the states of this automaton correspond to the reachable markings of N.

The language L accepted by such an automaton will be referred to as the *sequential behavior* of N. The words (sequences) of this language can be viewed as sequential observations of the behavior of N,

[1] Note that a C/E-system [Rei85] (or an Elementary Net system [Roz86]) is more restrictive than a one-safe P/T-net since the requirements for the firing of a transition t in a C/E-system are $({}^\cdot t \subseteq M) \wedge (t^\cdot \cap M = \emptyset)$.

i.e. observations made by observers able to see only a single event occurrence at a time. The ordering of symbols in these words reflects not only the (objective) causal ordering of event occurrences (transitions of the net), but also a (subjective) observational ordering resulting from a specific view of concurrent actions: whenever there are concurrent transitions in the net N, the corresponding automaton introduces an "artificial" nondeterminism whose nondeterministic choices correspond to the possible interleavings of these concurrent transitions. Therefore, the structure of such an automaton alone does not make it possible to decide whether the difference in ordering is caused by a conflict resolution (a nondeterministic decision), or by different observations of concurrency (interleavings). In order to extract the causal ordering of event occurrences, we will use the notion of *trace* [Maz86].

4 Traces

First, we define the notion of *concurrent alphabet*.

Definition 4.1 *A concurrent alphabet is a pair $\Sigma = (A, D)$ where A is a finite set of actions, called the alphabet of Σ, and D is a binary, symmetrical and reflexive, relation in A, called the* dependency *in Σ.*

$D(A) = (A, A^2)$ stands for the concurrent alphabet of total dependency on A, and $I_\Sigma = A^2 \setminus D$ stands for the *independency* in Σ.

Definition 4.2 *Let Σ be a concurrent alphabet; A^* represents the set of all finite sequences (words) of symbols in A, $\cdot$ stands for the concatenation operation, and the empty word is noted ε. We define the relation $\equiv_\Sigma$ as the least congruence in the monoid $[A^*; \cdot, \varepsilon]$ such that $(a, b) \in I_\Sigma \Rightarrow ab \equiv_\Sigma ba$.*

The relation $\equiv_\Sigma$ is referred to as the *trace equivalence over* Σ.

Definition 4.3 *Equivalence classes of $\equiv_\Sigma$ are called* traces over Σ.

A trace characterized by a word w and a concurrent alphabet Σ is denoted by $[w]_\Sigma$.

Thus a trace over a concurrent alphabet $\Sigma = (A, D)$ represents a set of words defined over A only differing by the order of adjacent symbols which are independent according to D. For instance, if a and b are two symbols of A which are independent according to D, the trace $[ab]_\Sigma$ represents the two words ab and ba. A trace is an equivalence class of words. A *trace language* is a set of traces over a given concurrent alphabet.

Let us return to the one-safe P/T-net N. We define the *dependency* in N as the relation $D_N \subseteq T \times T$ such that:

$$(t_1, t_2) \in D_N \Leftrightarrow {}^\cdot t_1 \cap {}^\cdot t_2 \neq \emptyset. \tag{1}$$

The complement of D_N is called the *independency* in N. If two independent actions occur next to each other in a firing sequence, the order of their occurrences is irrelevant (since they occur concurrently in this execution). Let $\Sigma_N = (T, D_N)$ be the concurrent alphabet associated with N and let L be the firing sequence language of N. We define the *trace behavior* of N as the set of equivalence classes of L defined by the relation $\equiv_{\Sigma_N}$. These equivalence classes are called *firing traces* of N. Such a class (trace) corresponds to a partial order (i.e. a set of causality relations) and represents all its linearizations (words).

To describe the behavior of a one-safe P/T-net by means of traces rather than sequences, we will need the dependency D_N of N (which can be deduced from the statical structure of N as defined by (1)) and *only one* linearization for each trace. Consider a language L' representing one arbitrary linearization (word) for each possible trace of the net. Let L' be such that

$$L = \bigcup_{w' \in L'} Pref(lin([w']_{\Sigma_N}))$$

where $lin([w]_{\Sigma_N})$ denotes the set of linearizations (words) of the trace (equivalence class) $[w]_{\Sigma_N}$ and $Pref(w)$ denotes the prefixes of w. Clearly, L' is a regular language. So, *the behavior of a one-safe*

P/T-net is fully characterized by the dependency D_N and an automaton which generates exactly L'. Let us call this automaton a *trace automaton* for N.

To construct such an automaton, we do not need to compute all the reachable markings of N: whenever several independent transitions are firable, we fire only one of these transitions in order to generate only one interleaving (linearization) of these transitions.

In the next section, we present an algorithm to construct a trace automaton for a given contact-free one-safe P/T-net N.

5 Constructing the Trace Automaton

The algorithm presented in Figure 1 is a classical depth-first search of the reachable markings of the net N with some important modifications.

The algorithm uses a *Stack* to hold the configurations that remain to be examined. Each configuration is composed by a marking M and two additional information: a "*Sleep* set" and a "$NDinfo$". A *Sleep* set is a subset of M-firable transitions. A "sleeping" transition will never be fired in the remainder of the search starting from the current marking M. Thus *Sleep* denotes a set of transitions which are firable but which will not be fired. A $NDinfo$ is an information which identifies the nondeterministic branch of the current marking M. Different possibilities for solving a nondeterminism lead to different nondeterministic branches. The root of all nondeterministic branches is the initial marking. A (hash) table H is also used to store the states of the automaton that have been explored. As usual, these states will correspond to reachable markings of the net N. A *Sleep* set, a $NDinfo$ and a "*succ*" set are associated to each state. The *succ* set contains the transitions leaving that state in the trace automaton. The states reachable from a given state of the trace automaton are obtained by firing, in the net, the transitions of the *succ* set associated with that state.

Since we suppose that N is contact-free, we do not have to check the requirement $t^{\cdot} \cap (M \setminus {}^{\cdot}t) = \emptyset$ before firing a transition t from a marking M. Dealing with contact-free one-safe P/T-nets is not a restriction. Indeed for every one-safe P/T-net there exists an equivalent contact-free one-safe P/T-net (i.e. a net that has the same firing sequence language) [Rei85]. Transitions $t_1, t_2, \ldots, t_n$ are referred to as being *in conflict* iff $({}^{\cdot}t_1 \cap {}^{\cdot}t_2 \cap \ldots \cap {}^{\cdot}t_n) \neq \emptyset$ (their occurrences are mutually exclusive and lead to different nondeterministic branches). A transition t that is not in conflict with another transition is *conflict-free*.

The first modification w.r.t. the classical algorithm is that we do not systematically fire all firable transitions from a given marking: we only choose some of them since we want to construct an automaton that generates only one interleaving whenever several independent transitions are firable. Our choices are motivated by the following two principles: we want to minimize the number of states of the trace automaton; we have to consider all possible behaviors of the net: each firing sequence of the net must be represented by some trace generated by the trace automaton.

Remember that each word w generated by the trace automaton defines a trace $[w]_{\Sigma_N}$. This means that all the linearizations of this trace and all the prefixes of these linearizations are firing sequences of the net N.

In order to minimize the number of states to be constructed, we define a priority scheme to choose amongst the firable transitions those that are to be fired. The highest priority (priority 1) is given to conflict-free transitions (these transitions will be fired one by one successively). Next, we give priority (priority 2) to transitions that are in conflict exclusively with firable transitions. When we fire such a transition, we also fire (from the current marking) the transitions that are in conflict with it in order to explore all possible nondeterministic cases (it corresponds to a branching in the trace automaton). All these possible cases lead to different nondeterministic branches (therefore, the $NDinfo$ of the markings obtained after firing these transitions are distinct). Finally, priority 3 is given to firable transitions that are in conflict with at least one nonfirable transition (this is a situation of *confusion* [Rei85]).

When there remain only firable transitions of priority 3, we proceed as follows. Let t be one of these firable transitions. We know that t is in conflict with at least one transition x that is not firable from

```
1. Initialize: Stack is empty; H is empty;
               enter (M0, ∅, 0) in H;
               push (M0, ∅, 0) into Stack
2. Loop: while Stack ≠ ∅ do
               begin
                 pop (M, Sleep, NDinfo) from Stack;
                 FT := CHOOSE-AMONGST({tf ∉ Sleep : ·tf ⊆ M}, Sleep, NDinfo)
                 for all tf ∈ FT: add tf to succ(M) in H
                 for all tf ∈ FT do
                     begin
                       nextM := M − ·tf + tf·;
                       NDinfo := tf->NDinfo; Sleep := tf->Sleep;
                       if nextM is already in H
                         then NDinfo-old := NDinfo associated with nextM in H;
                              if Same-ND-Branch(NDinfo-old,NDinfo)
                                then Sleep := Sleep ∪ succ(nextM);
                                     push (nextM, Sleep, NDinfo) into Stack
                                else Sleep-old := Sleep associated with nextM in H
                                     if Sleep-old ⊈ Sleep
                                       then enter (nextM, Sleep ∩ Sleep-old,
                                                                  NDinfo) in H;
                                            push (nextM, Sleep, NDinfo) into Stack
                         else enter (nextM, Sleep, NDinfo) in H;
                              push (nextM, Sleep, NDinfo) into Stack
                     end
               end
```

Figure 1: Algorithm

the current marking. But it is possible that x will become firable because of the evolution of some other tokens not in $\cdot t$. In that case, the firing of t could be replaced by the firing of x. In order not to miss this possibility, we consider several cases: we fire t from the current marking M, we fire the firable transitions $t_1, \ldots, t_n$ in conflict with t from M, if any, and then we "give the hand" to other tokens to see if they can make x firable (all these cases correspond to different nondeterministic branches). In the last case, since we have already considered the firing of t and $t_1, \ldots, t_n$ and since we are only interested by the "potential" firing of x, we do not consider $t, t_1, \ldots, t_n$ any longer and we put them in the *Sleep* set associated with the current configuration. If there still remain firable transitions of priority 3, we proceed again as described above with the new *Sleep* set, etc. In summary, when there remain only firable transitions of priority 3, all these transitions are fired from the current marking but with different *NDinfo* and *Sleep* sets. An example of such a situation is shown in Figure 2.a: from the current marking all the firable transitions have priority 3. The corresponding trace automaton is given below the net (the value of *Sleep* and *NDinfo* during the construction of the trace automaton is given between parentheses).

The function CHOOSE-AMONGST returns the next transition(s) to be fired according to the priority scheme. Moreover, this function associates two additional information to each chosen transition t_f to be fired: t_f->*Sleep* and t_f->*NDinfo* representing respectively the *Sleep* set and the *NDinfo* to be associated with the new marking obtained after firing t_f.

Another modification w.r.t. the classical depth-first search is that, if the current marking has already been reached, the search does not stop automatically. Indeed, suppose the current marking is M. The

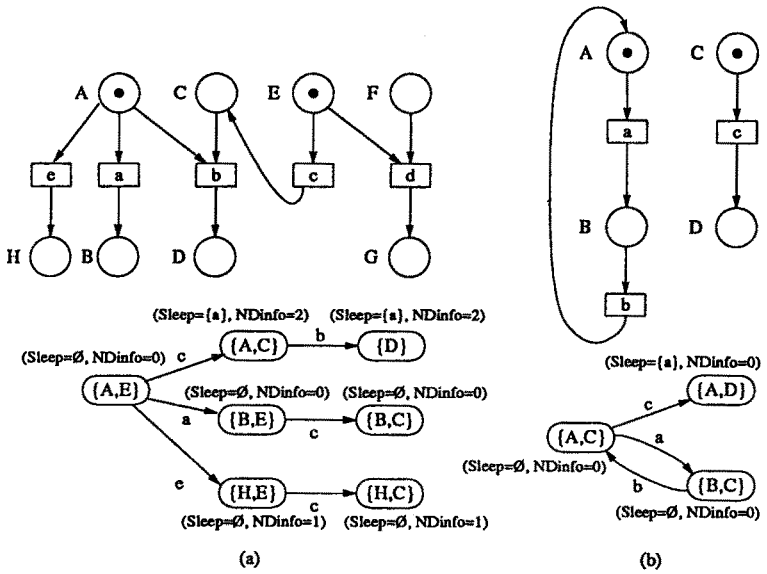

Figure 2: Nets and their corresponding trace automaton

search continues and after the firing of some transitions, the current marking becomes M again. In other words some of the tokens of M moved from their place and then returned back to their respective place. But it is possible that other tokens of M have not yet had the possibility of moving. In order to consider all possible behaviors, we have to "give the hand" to these tokens to see which other concurrent transitions could be fired (it is a kind of "fairness requirement" on the choices amongst independent firable transitions). This situation is illustrated in Figure 2.b: from the initial marking $\{A, C\}$, assume that we fire a and that next we fire b and go back to $\{A, C\}$; then we have to fire c although the marking $\{A, C\}$ has already been reached. This kind of situation can arise only with markings that can appear several times in the *same* nondeterministic branch of the trace automaton. Now, what happens if the next marking has already been reached in *another* nondeterministic branch ? The search from this next marking may stop if the Sleep set "*Sleep*-old" associated with this marking in the already reached marking table is included in the current Sleep set "*Sleep*". If this requirement is not satisfied, i.e. if there exists some transition t in *Sleep*-old (t is firable) such that t is not in *Sleep*, the search has to continue. Indeed, since t is not in *Sleep*, t (which is firable) ***has to be fired*** eventually and, since t is in *Sleep*-old, we know that t ***does not occur*** in the remainder of the search previously made starting from the next marking with *Sleep*-old as Sleep set.

Of course, there are many possible trace automata corresponding to a given contact-free one-safe P/T-net N. The algorithm described above constructs one of these possible automata. The order of the time complexity for our algorithm is given by the number of constructed transitions times the maximum number of simultaneous firable transitions. We do not claim that the trace automaton constructed by our algorithm is always the minimal one: it is often possible to further reduce the size of the trace automaton but at the cost of an increased time complexity.

Example 5.1 Let us consider the well-known dining philosophers problem. Figure 3 shows a net $dp2$ that represents two philosophers and their adjacent forks. The "classical" automaton, i.e. the one whose states correspond to all the reachable markings, and the trace automaton constructed by our algorithm that correspond to $dp2$ are presented in figure 4. The dotted part is not part of the trace automaton. ∎

The net $dp2$ is susceptible to deadlock: there is no firable transition from the marking $\{A_3, B_3\}$. Note

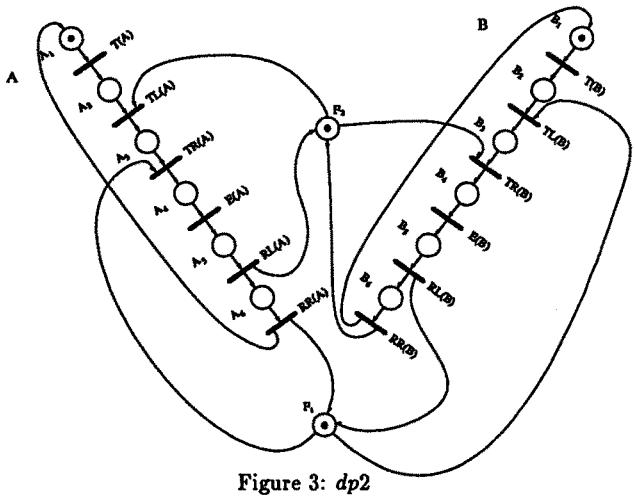

Figure 3: *dp2*

NET	Classical Automaton				Trace Automaton			
	Total Run Time (sec)	States	Trans.	Deadlock	Total Run Time (sec)	States	Trans.	Deadlock
dp2	0.76	21	34	13	0.7	13	14	13
dp3	3.64	99	240	25	1.1	25	28	20
dp4	43.52	465	1508	173	1.98	43	49	27
dp5	692.4	2163	8770	525	3.6	72	83	34

Table 1: Classical automaton versus trace automaton

that the state corresponding to the marking $\{A_3, B_3\}$ is a state of the trace automaton. Indeed, remember the main idea behind our algorithm is to choose some of the firable transitions to be fired whenever there are independent firable transitions. Thus, if there is *no* firable transition, we detect it (the deadlock-preserving property of partial order semantics was already pointed out in [Gai88], [Val88] among others).

Table 1 compares the performance of a depth-first search algorithm against the algorithm presented in this section for the nets *dp2* to *dp5* (five dining philosophers). The combinatorial explosion both in the number of states and transitions is clearly avoided by using trace automata. We also compare the run time needed to construct these automata (these results were obtained with a LISP prototype): our algorithm can be much faster than the classical one. Moreover, the deadlock is detected sooner: for *dp5*, the deadlock corresponds to the 34th reached state by our algorithm instead of the 525th one with a classical depth-first search.

Our method for modeling net behaviors has the potential of being *much more efficient, both in time and memory*, than the classical one. In the next sections, we show how to use trace automata to prove program correctness.

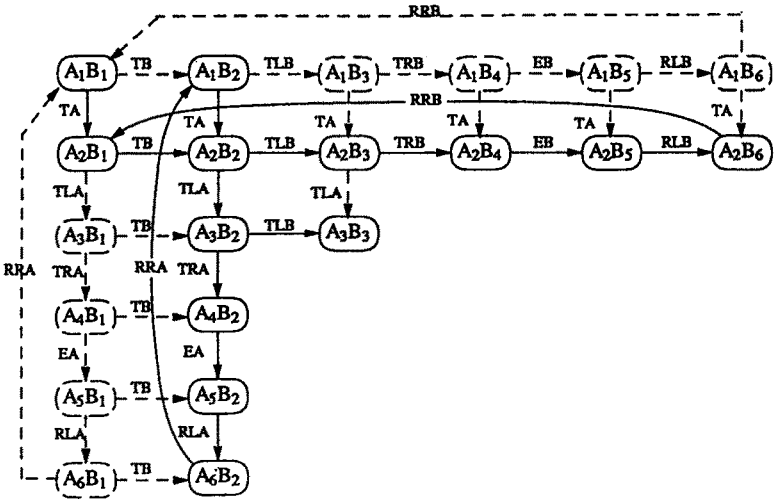

Figure 4: Classical automaton and trace automaton for *dp2*

6 Verification

Consider a set of processes P and a relation $\leq$ on this set such that $I \leq S$ iff I *refines* S (i.e., I is less nondeterministic than S). Let P be the set of all one-safe P/T-nets and $\leq$ the inclusion relation defined on the languages generated by these nets.

To verify that the implementation N_I effectively meets the specification N_S, the classical method consists in comparing the automata I and S respectively generating the firing sequence language of the nets N_I and N_S. If these automata are deterministic, verifying the language inclusion reduces to "simulating" I by S, i.e. checking that all that I can do can also be done by S. The cost of checking the existence of this simulation increases with the size of the automaton I, i.e. with the total number of reachable markings of the net N_I. Our claim is that this verification can be done at a lower cost, by using a trace automaton for N_I.

In the next section, we show how to express a criterion equivalent to language inclusion in terms of traces.

7 A Verification Criterion based on Traces

Let L_I (L_S) be the firing sequence language of a given one-safe P/T-net $N_I = (S_I, T_I, F_I, M_{0_I})$ (N_S). Let D_I (D_S) be the dependency in N_I (N_S) as defined previously and $\Sigma_I = (T_I, D_I)$ ($\Sigma_S = (T_S, D_S)$) the concurrent alphabet associated with N_I (N_S). We define $PO([w]_\Sigma)$ as the transitive closure of the relation $\{(a_i, a_j) : (a_i, a_j) \in D \text{ with } 1 \leq i < j \leq n\}$ if $w = a_1 a_2 \ldots a_n$ and D is the dependency of Σ. $PO([w]_\Sigma)$ represents the set of causality relations corresponding to $[w]_\Sigma$ ($PO([w]_\Sigma)$ is a partial order).

Let I' denote a trace automaton corresponding to N_I and let $L_{I'}$ be the language generated by I'.

Theorem 7.1 $lin([w]_{\Sigma_I}) \subseteq lin([w]_{\Sigma_S}) \Leftrightarrow PO([w]_{\Sigma_I}) \supseteq PO([w]_{\Sigma_S})$.

This theorem leads us directly to the following consideration: *to verify that the linearizations of a trace $[w']_{\Sigma_I}$ are included in the firing sequence language of a given one-safe P/T-net N_S, one needs to check if the word w' is a firing sequence of N_S and to verify that $PO([w']_{\Sigma_I}) \supseteq PO([w']_{\Sigma_S})$.*

Definition 7.1 *A one-safe P/T-net is called "restricted" if the following requirement is satisfied:*

$$\forall x, y \in T : ({}^\cdot x \cap {}^\cdot y) \neq \emptyset \Rightarrow (({}^\cdot x \cap {}^\cdot y) \not\subseteq (y^\cdot \cap x^\cdot)). \qquad (2)$$

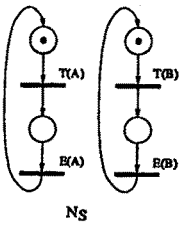

Figure 5: Specification for *dp2*

NET	Classical Simulation	Verification with P.O.
	Total Run Time (sec)	Total Run Time (sec)
dp2	0.96	0.74
dp3	5.08	1.18
dp4	60.1	2.36
dp5	940.4	4.58

Classical verification versus verification with P.O.

Theorem 7.2 *Let N_I be a one-safe P/T-net and N_S a "restricted" one-safe P/T-net. We have:*

$$L_I \subseteq L_S \Leftrightarrow \begin{cases} L_{I'} \subseteq L_S \\ \forall w' \in L_{I'} : PO([w']_{\Sigma_I}) \supseteq PO([w']_{\Sigma_S}) \end{cases}$$

Proof: Presented in the full paper. ∎

This theorem is of great interest because in most cases verifying the two inclusions of the second member can be done more efficiently than verifying the inclusion between all the linearizations. Indeed, the size of I' can be much smaller than the size of I, if N_I is a concurrent system.

8 The Verification Algorithm

We check that $L_{I'} \subseteq L_S$ as usual (see Section 6) by exploring the reachable states of $I' \times S$. A transition executed by I' must be simulated by S except if this transition is hidden. i.e. if this transition is not in the set T_S of transitions of the net N_S. Verifying the inclusion relation between the causality relations is done during the checking of the simulation of I' by S.

Algorithm: Presented in the full paper.

Example 8.1 Figure 5 shows a specification N_S for the two dining philosophers *dp2* of figure 3. ∎

The table of figure 5 compares the run time needed to verify that *dp2* is an implementation of N_S w.r.t. our criterion (theses results were obtained with a LISP prototype). In a similar way, this problem has been extended up to five philosophers and the corresponding results are presented in table 5: our algorithm can be much faster than the classical one. Moreover, from *dp3* and beyond, the memory required to store the states and the transitions due to the combinatorial explosion in the classical case is larger than the additional memory required to check inclusion between causality relations in our method.

9 Conclusions

Most of the work on partial orders deals with semantic problems or with algebraic properties and remains in a theoretical framework. There are only a few papers related to concurrent system verification using partial order models [Gai88][KP86][KP88][Pen90][PL89][PP90][PW84][Val89]. Our method has the advantages of being simple, fully algorithmic, and of extending directly conventional verification techniques. Moreover, this method has the potential of being much more efficient, both in time and memory, than the classical one. The dining philosophers example emphasizes clearly the power of our method: we avoid the part of the combinatorial explosion due to the modeling of concurrency by interleavings. The verification framework presented in this paper is restricted to nonlabeled P/T-nets and safety properties. Interesting future work would be to broaden the scope of our method to cope with labeled P/T-nets and liveness properties.

Acknowledgements

I would like to thank Professor Pierre Wolper for his enthousiastic supervision, for fruitful discussions and thoughtful hints. I am grateful to Jean-Yves Pirnay and Philippe Simar for helpful comments and contributing to the typesetting of this paper. My thanks are also addressed to Marianne Baudinet, Froduald Kabanza, François Schumacker and Dr. Wojciech Penczek for reading drafts of this paper.

References

[Gai88] H. Gaifman. Modeling concurrency by partial orders and nonlinear transition systems. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, pages 467–488, 1988.

[God89] P. Godefroid. Les modèles ordre partiel du parallélisme (partial order models for concurrency). Undergraduate thesis, Service d'Informatique, Université de Liège, June 1989.

[JK90] R. Janicki and M. Koutny. On some implementation of optimal simulations. To appear in *Proc. Computer-Aided Verification Workshop*, Rutgers, 1990.

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

[KP88] S. Katz and D. Peled. An efficient verification method for parallel and distributed programs. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, pages 489–507, 1988.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, 1978.

[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*, LNCS 255, pages 279–324, 1986.

[Maz88] A. Mazurkiewicz. Basic notions of trace theory. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, pages 285–363, 1988.

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

[PL89] D. K. Probst and H. F. Li. Abstract specification, composition and proof of correctness of delay-insensitive circuits and systems. Department of Computer Science, Concordia University, Montreal, Quebec Canada, 1989.

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

[PP90] D. Peled and A. Pnueli. Proving Partial Order Liveness Properties. ICALP, 1990.

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

[Rei85] W. Reisig. Petri nets: an introduction. EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1985.

[Roz86] G. Rozenberg. Behaviour of elementary net systems. In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course*, LNCS 254, pages 60–94, 1986.

[Val88] A. Valmari. Error detection by reduced reachability graph detection. In *Proc. 9th International Conference on Application and Theory of Petri Nets*, pages 95–112, Venice, 1988.

[Val89] A. Valmari. Stubborn sets for reduced state space generation. In *Proc. 10th International Conference on Application and Theory of Petri Nets*, vol. 2, pages 1–22, Bonn, 1989.

[Win86] G. Winskel. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course*, LNCS 255, pages 325–392, 1986.

[Wol89] P. Wolper. On the relation of programs and computations to models of temporal logic. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proc. Temporal Logic in Specification*, LNCS 398, pages 75–123, 1989.

[Zie80] W. Zielonka. Proving assertions about parallel programs by means of traces. ICS PAS Report 424, Institute of Computer Science, Polish Academy of Sciences, 1980.