Appears in: Information and Computation, Vol. 110, No. 2, Academic Press, May, 1994, pp 305-326.

A Partial Approach to Model Checking*

Patrice Godefroid Pierre Wolper
Université de Liège
Institut Montefiore, B28
4000 Liège Sart-Tilman, Belgium
Email: {god,pw}@montefiore.ulg.ac.be

Abstract

This paper presents a model-checking method for linear-time temporal logic that can avoid most of the state explosion due to the modelling of concurrency by interleaving. The method relies on the concept of Mazurkiewicz's trace as a semantic basis and uses automata-theoretic techniques, including automata that operate on words of ordinality higher than ω.

1 Introduction

Model checking [CES86, LP85, QS81, VW86] is an effective and simple method for verifying that a concurrent program satisfies a temporal logic formula. It works on finite-state programs and proceeds by viewing the program as a structure for interpreting temporal logic and by evaluating the formula on that structure. It is much simpler than temporal deductive proofs and can be easily and effectively implemented.

It has been intensively studied for linear-time temporal logic [LP85, VW86, Var89], branching-time temporal logic [CES86, EL85b, EL85a, Bro86] and temporal μ-calculi [EL86, Var88, Cle90, SW89]. It has been extended to probabilistic [Var85, PZ86, VW86, CY90] as well as real-time programs and logics [ACD90, AH90, HLP90]. It has been adapted to programs containing arbitrary numbers of identical processes [CGB86, CG87, GS87, WL89, KM89]. Methods for making it applicable to very large systems have been investigated [BCM$^+$90, CMB90, CVWY90, GS90]. Moreover, the results from its experimental use have been very encouraging [RRSV87, BCD85]. What more can be said about it?

In spite of all its success, almost all work around model checking is based on a very wasteful idea: modelling concurrency by interleaving. Even if one is not inclined to loose sleep about

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

whether interleaving semantics are adequate for concurrency, it remains unarguably silly to investigate the concurrent execution of n events by exploring all $n!$ interleavings of these events!

In this paper, we develop a simple method for applying model checking without incurring most of the cost of modelling concurrency by interleaving. Our method yields results identical to those of methods based on interleaving semantics, it just avoids most of the associated combinatorial explosion. It is quite orthogonal to model checking based on partial-order logics [PW84, KP86, Pen90]. Indeed, these logics are designed to be semantically more powerful. We are "only" more efficient. The idea that the cost of modelling concurrency by interleaving can be avoided in finite-state verification already appears in [PL90, Val91, Val90, God90]. We build upon this earlier work, specifically that of [God90], and bring to it the full capabilities of model checking.

We study model checking for linear-time temporal logic and adopt the automata-theoretic approach of [VW86, Var88, Wol89]. In this approach, the program is viewed as a collection of communicating automata on infinite words [Büc62]. It can thus include arbitrary fairness conditions. The negation of the formula to be checked is then also converted to an automaton on infinite words and the verification can be done by simply checking that the product of the automata describing the program and the automaton corresponding to the negation of the formula is nonempty. This is traditionally done by computing the product automaton which is where the cost of modelling concurrency by interleaving has to be paid.

In [God90] it is shown that the global behavior of a set of communicating processes can be represented by an automaton which can be much smaller than the usual product automaton. The basic idea is to build an automaton that only accepts one interleaving of each concurrent execution. The method is justified by using partial-order semantics, namely the concept of Mazurkiewicz's trace [Maz86] and the automaton is thus called a *trace automaton*. A trace automaton can be viewed as an automaton accepting at least one, but usually no more than one, interleaving for each *trace* (concurrent computation) of the concurrent program. Thus, together with the independence relation on transitions, this automaton fully represents the concurrent executions of the program. The practical benefit is that this automaton can be much smaller than the automaton representing all interleavings.

The motivating idea behind the method presented here is that, in the automata-theoretic approach to model checking, the trace automaton could be used in place of the product automaton. Unfortunately, this is not directly the case. However, we are able to obtain such a result by using a new type of automaton.

We consider automata operating on infinite words of ordinality higher than ω. Precisely, we define automata operating on words of length $\omega \times n$, $n \in \omega$.[1] We study these automata

[1]Interestingly, a related type of automata on ordinals was used by Büchi [Büc65b, Büc65a] to study the decidability of the monadic theory of the ordinals.

and show that their emptiness can be efficiently decided. We then show how, when it is viewed as an $\omega \times n$-automaton, the trace automaton can be used to improve the efficiency of model checking.

Finally, we conclude the paper with a comparison between our contributions and related work.

2 Automata and Model Checking

We briefly recall the essential elements of the automata-theoretic approach to model checking. More details can be found in [VW86, Wol89, ACW90] and in Chapter 4 of [Tha89]. The problem we consider is the following. We are given a concurrent program P composed of n processes P_i, each described by a finite automaton A_i on countably infinite words over an alphabet Σ_i. We are also given a linear-time propositional temporal logic formula f. The model-checking problem is then to verify that all infinite behaviors of the program P satisfy the temporal formula f.

The automata we use for describing the processes P_i are generalized Büchi automata[2], i.e. tuples $A = (\Sigma, S, \Delta, s_0, \mathcal{F})$, where

- Σ is a finite alphabet,
- S is a finite set of states,
- $\Delta \subseteq S \times \Sigma \times S$ is a transition relation,
- $s_0 \in S$ is the starting state, and
- $\mathcal{F} = \{F_1, \ldots, F_k\} \subseteq 2^S$ is a set of sets of accepting states.

Generalized Büchi automata are used to define languages of ω-words, i.e. functions from the ordinal ω to the alphabet Σ. Intuitively, a word is accepted by a Generalized Büchi automaton if the automaton has an infinite execution that intersects infinitely often each of the sets $F_j \in \mathcal{F}$.

Formally, we define the concept of a *computation* of A over an ω-word, i.e. a function from the ordinal ω to the alphabet Σ. A computation σ of A over an ω-word $w = a_1 a_2 \ldots$ is an ω-sequence $\sigma = s_0, s_1, \ldots$ (i.e. a function from ω to S) where $(s_{i-1}, a_i, s_i) \in \Delta$, for all $i \geq 1$. A computation $\sigma = s_0, s_1, \ldots$ is *accepting* if, for each $F_j \in \mathcal{F}$, there is some state in F_j that repeats infinitely often, i.e. for some $s \in F_j$ there are infinitely many $i \in \omega$ such that $s_i = s$. The ω-word w is *accepted* by A if there is an accepting computation of A over w. The set of ω-words accepted by A is denoted $L_\omega(A)$.

[2] Generalized Büchi automata differ from Büchi automata [Büc62] in that they have a set of sets of accepting states rather than just one set of accepting states.

An automaton A_P representing the joint behavior of the processes P_i can be computed by taking the product of the automata describing each process, actions that appear in several processes are synchronized, others are interleaved. Formally, the product ($\times$) of two (generalization to the product of n automata is immediate) generalized Büchi automata $A_1 = (\Sigma_1, S_1, \Delta_1, s_{01}, \mathcal{F}_1)$ and $A_2 = (\Sigma_2,\ S_2, \Delta_2, s_{02}, \mathcal{F}_2)$ is the automaton $A = (\Sigma, S, \Delta, s_0, \mathcal{F})$ defined by

- $\Sigma = \Sigma_1 \cup \Sigma_2$,
- $S = S_1 \times S_2$, $s_0 = (s_{01}, s_{02})$,
- $\mathcal{F} = \bigcup_{F_j \in \mathcal{F}_1} \{F_j \times S_2\} \cup \bigcup_{F_j \in \mathcal{F}_2} \{S_1 \times F_j\}$
- $((s,t), a, (u,v)) \in \Delta$ when
  - $a \in \Sigma_1 \cap \Sigma_2$ and $(s,a,u) \in \Delta_1$ and $(t,a,v) \in \Delta_2$,
  - $a \in \Sigma_1 \setminus \Sigma_2$ and $(s,a,u) \in \Delta_1$ and $v = t$,
  - $a \in \Sigma_2 \setminus \Sigma_1$ and $u = s$ and $(t,a,v) \in \Delta_2$.

Note that with this definition, the product automaton can have an infinite accepting computation that corresponds to a finite computation of some (but not all) of its components. Indeed, if a component i has a state s such that $s \in F_j$ for all $F_j \in \mathcal{F}_i$, then an infinite computation of the product in which component i stays indefinitely in state s will appear as accepting. This is a counterintuitive consequence of the straightforward definition we have chosen for the product. To avoid this, we adopt the following restriction on the acceptance conditions of the generalized Büchi automata we will use.

- either the acceptance condition is vacuous ($\mathcal{F} = \emptyset$), in which case the automaton can have either finite or infinite computations, or
- the set $\mathcal{F}$ contains at least two disjoint components, in which case the product automaton cannot have an accepting computation corresponding to a finite computation of the automaton.

For a given generalized Büchi automaton, it is quite straightforward to construct an equivalent automaton that satisfies this restriction. In programming terms, the restriction is a form of fairness condition imposed on the processes with nonvacuous acceptance conditions: their executions must be infinite (executions that might legitimately not be infinite can be modelled by using an additional "idling" action).

To obtain a model-checking procedure, the only fact we need about linear-time temporal logic is that, for each formula f, it is possible to build a generalized Büchi automaton A_f

that accepts exactly the infinite words satisfying the temporal formula f (the alphabet of this automaton is 2^P where P is the set of propositions appearing in the formula f) [WVS83, VW86, Wol89]. This construction is exponential in the length of the formula, but this is usually not a problem since the formulas to be checked are quite short and since the algorithm often behaves much better than its upper bound. The model-checking procedure is then the following:

1. Build the finite-automaton on infinite words for the *negation* of the formula f (one uses the negation of the formula as this yields a more efficient algorithm). The resulting automaton is $A_{\neg f}$.

2. Compute the product $A_G = \prod_{1 \leq i \leq n} A_i \times A_{\neg f}$ (in practice only the reachable states of this product).

3. Check if the automaton A_G is nonempty.

To check if the automaton A_G is nonempty, it is sufficient to check that its graph contains a strongly connected component that is reachable from the initial state and that includes a state from each of the sets F_j of its set $\mathcal{F}$ of accepting sets. This can be done with a linear-time algorithm [AHU74]. The complexity of this model-checking method is thus determined by the size of A_G. Note that model checking is often said to be of complexity "linear in the size of the program" which is correct if one measures the size of the program as the size of $\prod_{1 \leq i \leq n} A_i$. In practice, the limits of all model-checking methods come from the often excessive size of this product. The frustrating fact is that a lot of this excessive size is unnecessary: it is due to the modelling of concurrency by interleaving. This is what we are tempting to eliminate. Let us therefore turn to partial-order semantics.

3 Partial-Order Semantics and Trace Automata

In partial-order semantics, the possible behaviors of a concurrent system are described in terms of partial orders instead of sequences. More precisely, we use Mazurkiewicz's traces [Maz86] as semantic model. We briefly recall some basic notions of Mazurkiewicz's trace theory.

Definition 3.1 *A concurrent alphabet is a pair $\Sigma = (A, D)$ where A is a finite set of symbols, called the alphabet of Σ, and where D is a binary, symmetrical, and reflexive relation on A called the* dependency *in Σ.*

$I_\Sigma = A^2 \setminus D$ stands for the *independency* in Σ.

Definition 3.2 *Let $\Sigma = (A, D)$ be a concurrent alphabet, let A^* represent the set of all finite sequences (words) of symbols in A, let $\cdot$ stand for the concatenation operation, and let ε denote* *the empty word. 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 3.3 *Equivalence classes of $\equiv_\Sigma$ are called* traces over Σ*.*

The 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 that only differ 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.

Let us now return to a concurrent program described as the composition of n finite-state transition systems A_i and of a property f represented by the automaton $A_{\neg f}$. From now on, $A_{\neg f}$ will be denoted by A_{n+1}. Let $\Delta \subseteq S \times \Sigma \times S$ denote the transition relation of the product A_G of these automata.

For each transition $t = (\mathbf{s}, a, \mathbf{s}') \in \Delta$ with $\mathbf{s} = (s_1, s_2, \ldots, s_{n+1})$ and $\mathbf{s}' = (s'_1, s'_2, \ldots, s'_{n+1})$, the sets (by extension, we consider the states of A_G as sets in the following definitions[3])

- $^\bullet t = \{s_i \in \mathbf{s} : (s_i, a, s'_i) \in \Delta_i\}$
- $t^\bullet = \{s'_i \in \mathbf{s}' : (s_i, a, s'_i) \in \Delta_i\}$
- $^\bullet t^\bullet = {}^\bullet t \cup t^\bullet$

are called respectively the *preset*, the *postset* and the *proximity* of the transition t. Intuitively, the *preset*, resp. the *postset*, of a transition $t = (\mathbf{s}, a, \mathbf{s}')$ of A_G represents the states of the A_i's that synchronize together on a, respectively *before* and *after* this transition. We say that the A_i's with a nonempty preset and postset for a transition t are *active* for this transition.

Two transitions $t_1 = (\mathbf{s}_1, a_1, \mathbf{s}'_1)$, $t_2 = (\mathbf{s}_2, a_2, \mathbf{s}'_2) \in \Delta$ are said to be equivalent (notation $\equiv$) iff

$$^\bullet t_1 = {}^\bullet t_2 \wedge t_1^\bullet = t_2^\bullet \wedge a_1 = a_2.$$

Intuitively, two equivalent transitions represent the same transition but correspond to distinct occurrences of this transition. These occurrences can only differ by the states of the A_i's that are not active for the transition. We denote by T the set of equivalence classes defined over Δ by $\equiv$. By extension, we define the preset, resp. the postset, of an element of set T as being the

[3]We assume that the sets $S_1, \ldots, S_{n+1}$ (where S_i is the set of states of A_i) are pairwise disjoint.

preset, resp. the postset, of all transitions in the corresponding equivalence class. From now on, "transition" will refer to an element of T rather than of Δ.

We define the *dependency* in A_G as the relation $D_{A_G} \subseteq T \times T$ such that:

$$(t_1, t_2) \in D_{A_G} \Leftrightarrow {}^\bullet t_1^\bullet \cap {}^\bullet t_2^\bullet \neq \emptyset.$$

The complement of D_{A_G} is called the *independency* in A_G. If two independent transitions occur next to each other in a computation, the order of their occurrences is irrelevant, since they occur concurrently in this execution. (Note that there are other possible ways of defining the notion of dependency [GP93].)

Let $\Sigma_{A_G} = (T, D_{A_G})$ be the concurrent alphabet associated with A_G and let $L(A_G)$ be the language of finite words over T accepted by A_G (all states of A_G considered accepting). In other words, $L(A_G)$ is the set of finite sequences of transitions that the system A_G can perform from its initial state. We define the *trace behavior* of A_G as the set of equivalence classes of $L(A_G)$ defined by the relation $\equiv_{\Sigma_{A_G}}$. These equivalence classes are called *traces* of A_G. 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_G by means of traces rather than sequences, we need the dependency D_{A_G} of A_G and *only one* linearization for each trace of A_G. So, *the behavior of A_G is fully characterized by the dependency D_{A_G} and an automaton which generates (at least) one linearization for each trace*. We call such an automaton a *trace automaton* (denoted A_T) for A_G [God90].

Formally, the language $L(A_T)$ accepted by a trace automaton A_T satisfies the following relation:

$$L(A_G) = \bigcup_{w \in L(A_T)} Pref(lin([w]_{\Sigma_{A_G}})) \qquad (1)$$

where $lin([w]_{\Sigma_{A_G}})$ denotes the set of linearizations (words) of the trace (equivalence class) $[w]_{\Sigma_{A_G}}$ and $Pref(w)$ denotes the prefixes of w.

In [God90] an algorithm for constructing a trace automaton corresponding to a concurrent program[4] is given. To construct such an automaton A_T, we do not need to compute all the reachable states of A_G: whenever several independent transitions are executable, we execute only one of these transitions in order to generate only one interleaving (linearization) of these transitions. By construction, A_T is a "sub-automaton" of A_G (i.e. the states of A_T are states of A_G and the transitions of A_T are transitions of A_G). The order of the time complexity for the

[4]In [God90] a concurrent program is represented by a contact-free one-safe P/T-net instead of a parallel composition of sequential processes as defined here; since the former is a more general formalism (it allows the modelling of process creation/deletion) than the latter, the algorithm described in [God90] is still applicable in the context considered here.

algorithm presented in [God90] is given by the number of transitions in A_T times the maximum number of simultaneous executable transitions. In practice it turns out that building A_T often requires *much less time and memory* than building A_G.

For instance, the behavior of a simple protocol like the 5-dining-philosophers problem (see [God90]) that would classically require the use of a state-graph A_G containing 2163 states and 8770 transitions can be represented by a trace automaton A_T containing only 72 states and 83 transitions.

4 Using Trace Automata for Model Checking

In order to use the results of Section 3 for doing model checking, we would like to be able to proceed as follows.

1. Build the finite-automaton on infinite words for the *negation* of the formula f. The resulting automaton is $A_{\neg f}$.
2. Compute the *trace automaton* A_T corresponding to the concurrent executions of the processes A_i, $1 \leq i \leq n$, and of the automaton $A_{\neg f}$.
3. Check if the automaton A_T is nonempty.

Unfortunately, this is incorrect. First, there is an obvious reason that makes this incorrect which is that the trace automaton A_T is not defined as an automaton on infinite words and hence does not have a set $\mathcal{F}$. However, this problem can be easily solved. Let S_G and S_T respectively be the set of states of A_G and A_T. By construction, $S_T \subseteq S_G$. Let $\mathcal{F}_G = \{F_1, \ldots, F_k\}$ be the set of sets of accepting states of A_G. The set $\mathcal{F}_T$ of sets of accepting states of A_T is then defined by $\mathcal{F}_T = \{F'_1, \ldots, F'_k\}$ with $F'_i = F_i \cap S_T$.

Even if we extend the definition of A_T to include the set $\mathcal{F}_T$ defined above (let us call the result A_T^∞), we still cannot use A_T^∞ for model checking. Indeed it is quite possible that the automaton A_G obtained by the traditional computation of the product accepts some infinite word whereas A_T^∞ does not accept any infinite word. This might seem counter intuitive because one could expect that, if A_G accepts some word w, then by permuting independent transitions of the computation accepting w, one would obtain an accepting computation of A_T^∞ which would then be nonempty. This is actually true for finite computations but not for infinite computations. Indeed, consider two processes that are totally independent (their alphabets are completely disjoint). The trace automaton for these two processes can be one that allows any number of transitions of the first process *followed* by any number of transitions of the second process. This is is fine for finite computations, but for infinite computations, one will be left with either an infinite computation of the first process or one of the second process, but not an infinite computation of *both* processes. One can summarize this by saying that A_T^∞ represents the infinite computations of all processes, but not the joint infinite computations of unsynchronized processes. The following example illustrates this situation. Consider the generalized Büchi automata A and A' of Figures 1 and 2 where $\mathcal{F} = \{\{s_1\}, \{s_2\}\}$ and $\mathcal{F}' = \{\{s'_1\}, \{s'_2\}\}$ respectively. A possible trace automaton A_T^∞ is given in Figure 3. Its set of sets of accepting states is defined by $\mathcal{F}_T = \{\{(s_1, s'_0), (s_1, s'_1), (s_1, s'_2)\}, \{(s_2, s'_0)\}, \{(s_1, s'_1)\}, \{(s_1, s'_2)\}\}$. This automaton does not accept any word whereas there is a joint infinite execution of the automata A and A' that would be accepted by the corresponding global automaton.

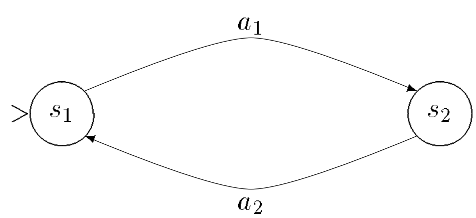

Figure 1: Generalized Büchi automaton A

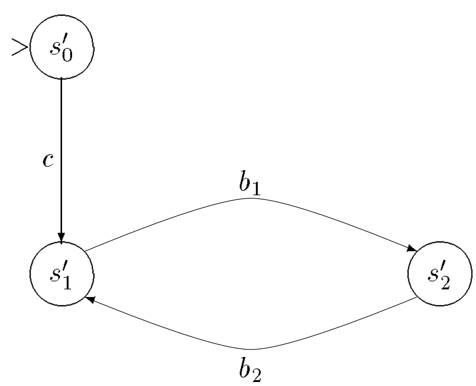

Figure 2: Generalized Büchi automaton A'

We now formalize the above discussion. Let A_G and A_T^∞ be respectively the product automaton and the trace automaton obtained by composing the generalized Büchi automata A_i, $1 \leq i \leq n+1$. Consider a computation of A_G or A_T^∞ on an infinite word w. One can view this

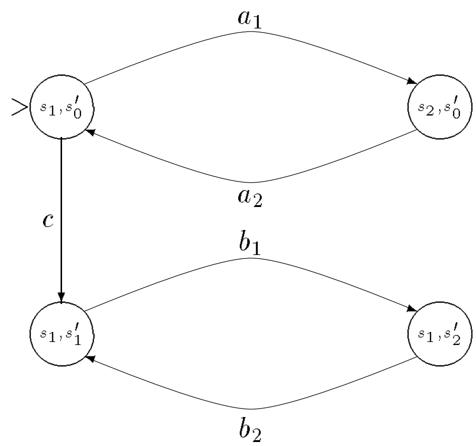

Figure 3: Trace automaton A_T^∞

computation as an infinite sequence of *transitions*, i.e., elements of set T defined in Section 3. For any transition of A_G or A_T^∞, one can identify the automata A_i that are *active* (as defined in Section 3) for this transition. This enables us to define the restriction of a computation of A_G or A_T^∞ to one of the components A_i.

Definition 4.1 *Given a trace or product automaton A obtained by composing the generalized Büchi automata A_i, $1 \leq i \leq n+1$, the restriction of a computation κ of A to the automaton A_i (denoted $\kappa|A_i$) is the subsequence of κ that contains only the transitions for which A_i is active.*

Note that the restriction of an infinite computation of A_G or A_T^∞ to an automaton A_i can be finite. We have the following.

Theorem 4.1 *Let κ be a computation (finite or ω-infinite) of the global automaton A_G obtained by composing the automata A_i, $1 \leq i \leq n+1$. Then, for every A_i, there is a computation κ_i (finite or ω-infinite) of the trace automaton A_T^∞ such that $\kappa|A_i = \kappa_i|A_i$.*

Proof: Consider a computation κ of A_G. Consider then the computation κ_i of A_T^∞ which has the longest restriction $\kappa_i|A_i$ that is a prefix of $\kappa|A_i$ (if there are several such computations, choose one of these arbitrarily). Let t_i be the first transition of A_i in κ that is not in this prefix. If there is no such transition, the theorem holds. Else, let us consider the prefix of κ that ends with transition t_i. This finite computation is then the prefix of a trace of which, by definition of A_T, at least one linearization is generated by A_T. The projection on A_i of any of these linearizations is $\kappa_i|A_i \cdot t_i$, which is longer than what we have assumed to be the longest projection on A_i of a computation of A_T^∞ that is a prefix of $\kappa|A_i$. Since computations of A_T are trivially computations of A_T^∞, we have a contradiction and the theorem follows. ∎

Note that it is not true that there is a single computation κ' of A_T^∞ such that $\kappa|A_i = \kappa'|A_i$ for all A_i's. In spite of this, Theorem 4.1 lets us obtain an interesting result, namely that the trace automaton can be used for model checking in cases where only one of the components is required to have an infinite computation. This is the case if all but one of the automata A_i have a vacuous accepting condition, i.e. have an empty set $\mathcal{F}$. This is proved in the following theorem.

Theorem 4.2 *Let A_i, $1 \leq i \leq n+1$ be generalized Büchi automata all but one of which have a vacuous accepting condition. Let A_G and A_T^∞ be the product and trace automata obtained by composing the automata A_i. Then, the automaton A_G is nonempty (has at least one infinite accepting computation) iff the trace automaton A_T^∞ is nonempty.*

Proof: Assume A_G has an infinite accepting computation κ and let A_j be the generalized Büchi automaton that has a nonvacuous acceptance condition. From Theorem 4.1, we know that there is a computation κ_j of A_T^∞ such that $\kappa|A_j = \kappa_j|A_j$. Since we have assumed in Section 2 that the product automaton A_G cannot have an accepting computation corresponding to a finite computation of A_j, $\kappa|A_j$, and hence also $\kappa_j|A_j$, are infinite. Moreover, $\kappa|A_j$ intersects infinitely often each of the sets $F \in \mathcal{F}_j$ and, given that $\kappa_j|A_j = \kappa|A_j$, this is also the case for κ_j. This proves that A_T^∞ is nonempty.
The other direction of the theorem is directly obtained from the immediate fact that all computations of the trace automaton are also computations of the global automaton. ∎

In practice, Theorem 4.2 enables us to use the trace automaton for model checking in the cases where the program does not operate under some fairness hypothesis, or when the fairness hypothesis is incorporated into the formula to be verified. Indeed, in those circumstances, the automata representing the program will have vacuous accepting conditions and the automaton obtained from the formula to be checked will be the only one with a nonempty set $\mathcal{F}$.

5 Automata on $(\omega \times n)$-words

Trace automata do not adequately represent the ω-computations of the components from which they are built because infinite computations cannot be concatenated. Actually, with the help of a little abstraction, infinite computations could very well be concatenated. One can simply think of computations whose length is an ordinal larger than ω. Since we are only interested in the concatenation of a finite number of infinite computations we will only study computations of length $\omega \times n$ where $n \in \omega$. The definitions of Section 2 can be quite naturally extended to words and computations of length $\omega \times n$ (for other definitions of automata on ordinals, see [Büc65b, Büc65a]).

A word of length $\omega \times n$ over the alphabet Σ is a function w from the ordinal $\omega \times n$ to Σ. We use automata that are defined exactly as in Section 2 and simply change the definition of a computation. A *computation* of an automaton $A = (\Sigma, S, \Delta, s_0, \mathcal{F})$ on a word w of length $\omega \times n$ is a function σ from $\omega \times n$ to S that satisfies the following conditions:

1. $\sigma(0) = s_0$;
2. for each successor ordinal $\alpha + 1 \in \omega \times n$, $(\sigma(\alpha), w(\alpha), \sigma(\alpha + 1)) \in \Delta$;
3. for each limit ordinal $\lambda \in \omega \times n$, there is an infinite sequence of ordinals α whose limit is λ such that $\sigma(\alpha) = \sigma(\lambda)$.

The notions of accepting computation and accepted word are essentially unchanged. A computation σ is *accepting* if, for each $F_j \in \mathcal{F}$, there is some state in F_j that repeats infinitely often, i.e., for some $s \in F_j$ there are infinitely many $i \in \omega \times n$ such that $s_i = s$. The $\omega \times n$-word w is *accepted* by A if there is an accepting computation of A over w. The set of $\omega \times n$ words accepted by A is denoted $L_{\omega \times n}(A)$. Note that if an automaton accepts a word of length $\omega \times n$, $n \geq 1$, it also accepts a word of length $\omega \times n'$ for all $n \leq n' < \omega$.

Checking that $L_{\omega \times n}(A)$ is nonempty can be done by computing the maximal strongly connected components of A.

Theorem 5.1 *Let $A = (\Sigma, S, \Delta, s_0, \mathcal{F})$ be an automaton. Then, $L_{\omega \times n}(A) \neq \emptyset$ iff there is a sequence of nontrivial maximal strongly connected components $C_1, \ldots C_n$ in A such that*

- *C_1 is accessible from s_0 and C_{i+1} is accessible from C_i, for $1 \leq i < n$ and*
- *for each $F_j \in \mathcal{F}$, there is some C_i such that $F_j \cap C_i \neq \emptyset$.*

Proof: Assume that A has an accepting computation κ of length $\omega \times n$. Since A is finite state, the first ω-sequence of this computation must, from some point on, have all its states included in a nontrivial maximal strongly connected component C_1 of A. Similarly, the second ω-sequence must start in a state of C_1 and must end in a component C_2 accessible from C_1 (it could actually be C_1 itself). Repeating the same line of thought for all ω sequences in κ up to the nth, one concludes the existence of the sequence of maximal strongly connected components C_i, $1 \leq i \leq n$. Moreover, since κ is accepting, κ contains at least one state of each set $F_j \in \mathcal{F}$ infinitely often. And, since these states appear infinitely often, they must be in one of the components C_i. This proves that the condition given in the theorem is necessary.

To prove that it is sufficient, let us assume the existence of the sequence of connected components and construct an accepting computation. The first ω-sequence in the computation starts in the initial state, has a finite prefix that leads it to the component C_1 and then goes infinitely often through all states of C_1. The second ω-sequence starts from any state in C_1, has a finite prefix that leads to C_2 and then goes infinitely often through all states in C_2. The following ω-sequences in the computation up to the nth are defined similarly. Since it goes through all states of all components C_i infinitely often, this computation is clearly accepting. ∎

The interesting aspect of the definitions we have just given is that if we consider the trace automaton as an automaton on words of length $\omega \times n$, then it represents all infinite computations of the combined automata. To prove this, we first establish a lemma.

Lemma 5.2 *Let κ be a finite computation of the global automaton A_G obtained by composing the automata A_i, $1 \leq i \leq n+1$. Then, there is a finite computation κ' of A_T such that for all $1 \leq i \leq n+1$, $\kappa|A_i$ is a prefix of $\kappa'|A_i$.*

Proof: The lemma is a direct consequence of the definition of trace automata. Indeed, if κ is a finite computation of A_G, then there is a representative of a trace that extends κ that is a computation κ' of A_T. Thus, since adjacent transitions of a single process are never independent, we must have that $\kappa|A_i$ is a prefix of $\kappa'|A_i$. ∎

If we extend the notion of computation used in Section 4 to sequences of transitions of length $\omega \times n$, we have the following.

Theorem 5.3 *Assume that the global automaton A_G obtained by composing the automata A_i, $1 \leq i \leq n+1$ has an accepting ω-computation κ. Then, there is an accepting computation κ' of length at most $\omega \times (n+1)$ of the trace automaton A_T^∞.*

Proof: We use a pumping argument to prove this theorem. We start by considering a finite prefix κ_f of κ that is long enough to satisfy the following condition for each automaton A_i that has a nonvacuous accepting condition. We describe the condition for a generic automaton A_i. Let $\mathcal{F} = \{F_1, \ldots, F_k\}$ be the accepting condition of A_i. For each set F_j, $1 \leq j \leq k$, there is thus at least some state $s_j \in F_j$ that appears infinitely often in κ (more precisely, this state appears in κ as the component of the global state corresponding to A_i). This implies that by focusing on selected states of κ, one can identify infinitely often the sequence $s_1, s_2, \ldots, s_k$ (of A_i components of the global state). The condition is that the sequence $s_1, s_2, \ldots, s_k$ can be selected from κ_f at least as many times as there are states in A_T.

We then consider the computation κ_{Tf} of A_T^∞ that satisfies the condition given in Lemma 5.2 for κ_f. This computation thus also satisfies the condition we have imposed on κ_f. Moreover, since the sequence $s_1, s_2, \ldots, s_k$ appears at least as many times as there are states in A_T, each of the states s_j must appear at least twice as component of the same state of A_T. Thus κ_{Tf} contains, for instance, a subsequence that starts in a global state $\mathbf{s}$ whose A_i component is s_1, that goes through at least one state of each of the accepting sets F_j of A_i and that ends again in the state $\mathbf{s}$. We call such a subsequence of κ_{Tf} an A_i-complete subsequence. Note moreover that our assumption that each set $\mathcal{F}$ contains at least two components implies that A_i-complete subsequences are nontrivial (contain at least one transition). We have thus established that κ_{Tf} contains at least one A_i-complete subsequence for each of the automata A_i with nonvacuous accepting conditions.

The next step is to choose in κ_{Tf} an A_i-complete subsequence for each A_i with a nonvacuous accepting computation and to sort these by order of appearance of their first state. Let $\sigma_1, \sigma_2, \ldots, \sigma_\ell$, $\ell \leq n+1$, be the A_i-complete subsequences taken in this order and let $\mathbf{s}_1, \mathbf{s}_2, \ldots \mathbf{s}_\ell$ be their respective first states (these states thus appear in κ_{Tf} in that order) and let $\mathbf{s}_0$ be the first state of κ_{Tf}. Then, if we denote by $[\mathbf{s}_i, \mathbf{s}_j]$ the portion of κ_{Tf} that is included between $\mathbf{s}_i$ and $\mathbf{s}_j$ the following is an accepting computation of A_T^∞ of length $\omega \times \ell$ (w^ω represents an infinite repetition of the word w)

$$[\mathbf{s}_0, \mathbf{s}_1]\sigma_1^\omega[\mathbf{s}_1, \mathbf{s}_2]\sigma_2^\omega[\mathbf{s}_2, \mathbf{s}_3]\sigma_3^\omega \cdots [\mathbf{s}_{\ell-1}, \mathbf{s}_\ell]\sigma_\ell^\omega.$$

∎

To use the trace automaton for model checking, we also need the converse of Theorem 5.3. However, this does not hold in general since it requires that a computation of length $\omega \times (n+1)$ be merged into a computation of length ω which is not always possible since only independent transitions can be interchanged. More precisely, if A_T^∞ is empty, Theorem 5.3 guaranties that A_G is also empty and hence that the program satisfies the property. If A_T^∞ is nonempty and has a computation of length ω, A_G is also nonempty and the program does not satisfy the property. The difficult case is when A_T^∞ has an accepting computation of length greater than ω. A simple approach to deal with this situation is to reconstruct part of A_G in order to determine whether the computation of A_T^∞ that has been found is an artifact or actually corresponds to a computation of A_G. It might seem that reconstructing part of A_G looses the advantage of the partial order approach, but note that this need not be done in all cases, and that the construction is limited to the accepting computation of A_T^∞ that has been found. Concretely, one can do the partial construction of A_G from the projections on the various processes of the sequence of strongly connected components of A_T^∞ that defines an accepting computation.

The partial construction of A_G can be avoided in even more cases if one first checks whether the accepting computation of A_T^∞ satisfies a "separability" condition. Consider a computation of length $\omega \times (n+1)$. For each ω-sequence in this computation, i.e. part of the computation corresponding to an interval $[\omega \times j, \omega \times (j+1)[$, we define the *repeating part* of this ω-sequence as its suffix that only contains states that appear infinitely often. The rest of the ω-sequence is then its *finite prefix*. We call a computation *separable* if for all $0 \leq i < j \leq n$, all transitions in the repeating part of $[\omega \times i, \omega \times (i+1)[$ are independent of all transitions in the finite prefix of $[\omega \times j, \omega \times (j+1)[$. We can then show that the converse of Theorem 5.3 holds for separable computations.

Theorem 5.4 *Let A_i, $1 \leq i \leq n+1$ be generalized Büchi automata. Let A_G and A_T^∞ be the product and trace automata obtained by composing the automata A_i. Then, if the trace automaton A_T^∞ has at least one separable accepting computation of length at most $\omega \times (n+1)$, the automaton A_G is nonempty (has at least one accepting computation).*

Proof: Notice that if A_T^∞ has a separable accepting computation of length $\omega \times (n+1)$, it has an accepting computation of the form

$$\sigma_0 \sigma_{0r}^\omega \sigma_1 \sigma_{1r}^\omega \cdots \sigma_n \sigma_{nr}^\omega$$

where σ_i and σ_{ir} are finite computations and where all transitions in σ_{ir} are independent with respect to all transitions in σ_j for $0 \leq i < j \leq n$. As a consequence the following is an accepting ω-computation of A_G

$$\sigma_0 \sigma_1 \cdots \sigma_n (\sigma_{0r} \sigma_{1r} \cdots \sigma_{nr})^\omega.$$

∎

A sufficient condition for A_T^∞ to have a separable condition is that is has a sequence of strongly connected components as in the condition of Theorem 5.1 and furthermore that for all $1 \leq i < j \leq (n+1)$, the transitions appearing in C_i are independent from those appearing in the path from C_{j-1} to C_j. For instance, the trace automaton in Figure 3 has a separable accepting computation $(a_1a_2)^\omega c(b_1b_2)^\omega$ of length $\omega \times 2$.

In summary, the procedure for checking whether A_T^∞ has a computation corresponding to a computation of A_G is the following. We first determine if A_T^∞ has a sequence of strongly connected components that satisfy the condition of Theorem 5.1. If there is no such sequence, A_G is empty. If there is such sequence, we check whether it satisfies the separability condition above. If it does, A_G is nonempty. In the remaining cases, a partial search of A_G is required to obtain a definite answer. Finally, note that another possible approach would be to guaranty the existence of a separable computation of A_T^∞ whenever A_G has a computation by using a different construction of A_T. Indeed, in all above, the only property of trace automata we have used is property (1) given in Section 3. Specific constructions of trace automata often have additional properties and can be tailored to satisfy specific requirements.

6 Conclusions and Comparison with Other Work

The closest work to the one presented here is certainly that of Valmari [Val90]. His paper also addresses the problem of adapting to model checking a method that avoids considering all interleavings of independent events while generating the state space of a concurrent program. It is likewise based on linear-time temporal logic, but uses a different strategy from the one we presented here. In our approach, the fact that the order of actions that appear in the formula cannot be ignored while constructing the trace automaton is handled by treating the property as any other component of the concurrent program. In [Val90], the problem is solved by a less discriminating approach. Precisely, the use of the "next" temporal operator is disallowed and all transitions that can affect the truth value of any state predicate appearing in the formula are considered as dependent. Prohibiting "next" is indeed important in this approach since in the presence of this operator all transition could potentially affect the truth value of the formula and hence would have to be considered as dependent and this would annihilate any benefit coming from the use of a partial-order approach. In our paper, we do handle the full temporal logic, and, actually, we can also handle extended temporal logics like that of [Wol83]. However, it should be noted that our interpretation of "next" is different from the one that causes problems in the method used by Valmari: we interpret "next" as meaning "next action monitored by the formula" rather than "next state of the program".

The treatment of fairness properties is also an important difference between Valmari's approach and ours. In Valmari's approach, the only way to represent fairness conditions would be to incorporate them in the formula (which hence has the form $fair \supset property$) whereas we represent them as Büchi conditions on the processes. The interaction of fairness conditions and partial-order methods is problematic since a fairness condition often concerns all processes involved in the program and hence introduces many dependencies which can wipe out the benefit of the approach. Our solution is to represent fairness assumptions in a distributed way, by assigning progress conditions to individual processes whenever possible. The drawback of this strategy is that it does not yield naturally to the expression of some fairness constraints.

A final element of comparison is the algorithm for computing an automaton that only represents "some" interleavings of concurrent events that lies at the heart of both approaches. First, note that an important advantage of the use of $\omega \times n$ automata is that no modification of an algorithm suitable for finite computations is necessary. On the other hand, in [Val90] the algorithm has to be modified, which increases the size of the state space that is generated. Furthermore, the technical ideas behind the constructions used in both approaches differ. Valmari uses an algorithm based on "Stubborn Sets", we use the construction of the "Trace Automaton" given in [God90]. This difference also influences the effectiveness of the model-checking methods. However, this influence is not extremely clear cut and is orthogonal to that of the strategy being used. It is quite possible that for some problems the "Trace Automaton" algorithm is best whereas for others the "Stubborn Sets" one is preferable. It is worth noticing that parts of both algorithms can be combined in order to achieve better reductions [WG93].

How good really is our method? It is hard to give a precise answer since it might be no better than interleaving methods when there is very tight coupling between the processes and dramatically better when there is no coupling between the processes. In the latter case, we could claim as is done in [BCM+90] that we can check systems with astronomical numbers of (interleaving semantics) states. Of course this should be taken with a grain of salt since the fact that checking only part of this enormous state space is sufficient indicates that most of the interleaving-semantics states are uninteresting. In [BCM+90] a similar phenomenon occurs, the difference being that the verification of large systems is made possible not by ignoring an irrelevant part of their state space, but by computing with an efficient symbolic representation of sets of states and transition relations.

The construction of trace automata of [God90] has been implemented and shows promising results which bodes well for the method described in this paper. Other work on the implementation and use of trace-automata-like techniques has also appeared. In [GW91b], model checking restricted to safety properties is considered. Several alternative "partial-order" verification algorithms are presented in [HGP92] and their performance on real-protocols is evaluated. Further results on the practicality of using trace-automata-like constructions are presented in [GHP92].

Finally, note that our method has the advantages of "on the fly verification" [CVWY90, JJ89, BFH90, HPOG89]. By this we mean that we build the automaton for the combination of the program and property without ever building the automaton for the program. Maybe surprisingly, this automaton is often smaller than the automaton for the program alone because the property acts as a constraint on the behavior of the program. Our method thus has a head start over methods that require the state graph of the program to be built.

Acknowledgements

We wish to thank Jerry Burch, David Dill, Wolfgang Thomas, Frits Vaandrager, Antti Valmari and anonymous referees for helpful comments on this paper. A preliminary version of this paper appeared in the Proceedings of the 6th Symposium on Logic in Computer Science [GW91a].

References

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 414–425, Philadelphia, June 1990.

[ACW90] S. Aggarwal, C. Courcoubetis, and P. Wolper. Adding liveness properties to coupled finite-state machines. *ACM Transactions on Programming Languages and Systems*, 12(2):303–339, 1990.

[AH90] R. Alur and T. Henzinger. Real-time logics: complexity and expressiveness. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 390–401, Philadelphia, June 1990.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, Reading, 1974.

[BCD85] M. Browne, E.M. Clarke, and D.L. Dill. Automatic circuit verification using temporal logic: Two new examples. In *IEEE Int. Conf. on Computer Design: VLSI and Computers*, Port Chester, October 1985.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, June 1990.

[BFH90] A. Bouajjani, J. C. Fernandez, and N. Halbwachs. On the verification of safety properties. Technical Report SPECTRE L12, IMAG, Grenoble, March 1990.

[Bro86] Michael C. Browne. An improved algorithm for the automatic verification of finite state systems using temporal logic. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 260–266, Cambridge, June 1986.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method and Philos. Sci. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.

[Büc65a] J.R. Büchi. Decision methods in the theory of ordinals. *Bull of the AMS*, 71:767–770, 1965.

[Büc65b] J.R. Büchi. Transfinite automata recursions and weak second order theory of ordinals. In *Proc. Internat. Congr. Logic, Method and Philos. Sci. 1964*, pages 2–23, Amsterdam, 1965. North Holland.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CG87] E. M. Clarke and O. Grümberg. Avoiding the state explosion problem in temporal logic model-checking algorithms. In *Proc. 6th ACM Symposium on Principles of Distributed Computing*, pages 294–303, Vancouver, British Columbia, August 1987.

[CGB86] E. M. Clarke, O. Grümberg, and M. C. Browne. Reasoning about networks with many identical finite-state processes. In *Proc. 5th ACM Symposium on Principles of Distributed Computing*, pages 240–248, Calgary, Alberta, August 1986.

[Cle90] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27:725–747, 1990.

[CMB90] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagram. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 23–32, Rutgers, June 1990. Springer-Verlag.

[CVWY90] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242, Rutgers, June 1990.

[CY90] C. Courcoubetis and M. Yannakakis. Markov decision processes and regular events. In *Proc. 17th Int. Coll. on Automata Languages and Programming*, volume 443, pages 336–349, Coventry, July 1990. Lecture Notes in Computer Science, Springer-Verlag.

[EL85a] E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, January 1985.

[EL85b] E.A. Emerson and C.-L. Lei. Temporal model checking under generalized fairness constraints. In *Proc. 18th Hawaii International Conference on System Sciences*, Hawaii, 1985.

[EL86] E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 267–278, Cambridge, June 1986.

[GHP92] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *Proc. 4th Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 178–191, Montreal, June 1992. Springer-Verlag.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185, Rutgers, June 1990.

[GP93] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 438–449, Elounda, June 1993. Springer-Verlag.

[GS87] S. German and A.P. Sistla. Reasoning with many processes. In *Proc. of the First Symposium on Logic in Computer Science*, pages 138–152, Ithaca, June 1987.

[GS90] S. Graf and B. Steffen. Using interface specifications for compositional reduction. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 186–196, Rutgers, June 1990. Springer-Verlag.

[GW91a] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, pages 406–415, Amsterdam, July 1991.

[GW91b] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proc. 3rd Workshop on Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, pages 332–342, Aalborg, July 1991.

[HGP92] G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. 12th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, pages 349–363, Lake Buena Vista, Florida, June 1992. North-Holland.

[HLP90] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit-clock temporal logic. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 402–413, Philadelphia, June 1990.

[HPOG89] N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A.C. Glory. Specifying, programming and verifying real-time systems, using a synchronous declarative language. In *Workshop on automatic verification methods for finite state systems*, volume 407 of *Lecture Notes in Computer Science*, pages 213–231, Grenoble, June 1989.

[JJ89] C. Jard and T. Jeron. On-line model-checking for finite linear temporal logic specifications. In *Workshop on automatic verification methods for finite state systems*, volume 407 of *Lecture Notes in Computer Science*, pages 189–196, Grenoble, June 1989.

[KM89] R. P. Kurshan and K. McMillan. A structural induction theorem for processes. In *Proceedings of the Eigth ACM Symposium on Principles of Distributed Computing*, Edmonton, Alberta, August 1989.

[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.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[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.

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

[PL90] D. K. Probst and H. F. Li. Using partial-order semantics to avoid the state explosion problem in asynchronous systems. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 146–155, Rutgers, June 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.

[PZ86] A. Pnueli and L. Zuck. Probabilistic verification by tableaux. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[QS81] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proc. 5th Int'l Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, 1981.

[RRSV87] J.L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in xesar of the sliding window protocol. In *Proc. IFIP WG 6.1 7th Int. Conf. on Protocol Specification, Testing and Verification*, pages 235–250, Zurich, 1987. North Holland.

[SW89] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. In *Proc. 15th Col. on Trees in Algebra and Programming.* Lecture Notes in Computer Science, Springer-Verlag, 1989.

[Tha89] André Thayse and et al. *From Modal Logic to Deductive Databases: Introducing a Logic Based Approach to Artificial Intelligence.* Wiley, 1989.

[Val90] A. Valmari. A stubborn attack on state explosion. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165, Rutgers, June 1990.

[Val91] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515, Bonn, June 1991. Springer-Verlag.

[Var85] M. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proc. 26th IEEE Symp. on Foundations of Computer Science*, pages 327–338, Portland, October 1985.

[Var88] M. Vardi. A temporal fixpoint calculus. In *Proc. 15th ACM Symp. on Principles of Programming Languages*, pages 250–259, San Diego, January 1988.

[Var89] M. Vardi. Unified verification theory. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proc. Temporal Logic in Specification*, volume 398, pages 202–212. Lecture Notes in Computer Science, Springer-Verlag, 1989.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[WG93] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *Proc. CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 233–246, Hildesheim, August 1993. Springer-Verlag.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems, Proc. Int. Workshop, Grenoble*, volume 407, pages 68–80, Grenoble, June 1989. Lecture Notes in Computer Science, Springer-Verlag.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2):72–99, 1983.

[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*, volume 398 of *Lecture Notes in Computer Science*, pages 75–123, 1989.

[WVS83] P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *Proc. 24th IEEE Symposium on Foundations of Computer Science*, pages 185–194, Tucson, 1983.