A Partial Approach to Model Checking
(Extended Abstract)

Patrice Godefroid       Pierre Wolper
Institut Montefiore, B28
Université de Liège
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 avoids 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 $\omega$.

1 Introduction

Model Checking [13, 29, 35, 44] 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 [29, 44, 43] branching-time temporal logic [13, 19, 18, 6] and temporal $\mu$-calculi [20, 42, 14, 37]. It has been extended to probabilistic [41, 33, 44, 17] as well as real-time programs and logics [2, 3, 25]. It has been adapted to programs containing arbitrary numbers of identical processes [12, 11, 21, 47, 28]. Methods for making it applicable to very large systems have been investigated [10, 15, 16, 23]. Moreover, the results from its experimental use have been very encouraging [36, 5]. 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 lose sleep about 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 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 [32, 27, 31]. 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 [34, 39, 40, 22]. We build upon this earlier work, specifically that of [22], 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 [44, 42, 46]. In this approach, the program is viewed as a collection of communicating automata on infinite words [7]. 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 [22] 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
The automata we use for describing the processes \( P = (\Sigma, S, \Delta, s_0, \mathcal{F}) \), where \( \Sigma \) is an alphabet, \( S \) is a 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 \( \omega \)-words, i.e. functions from the ordinal \( \omega \) to the alphabet \( \Sigma \). 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 run of \( A \) over an \( \omega \)-word, i.e. a function from the ordinal \( \omega \) to the alphabet \( \Sigma \). A run \( \sigma \) of \( A \) over an \( \omega \)-word \( w = a_1 a_2 \ldots \) is an \( \omega \)-sequence \( \sigma = s_0, s_1, \ldots \) (i.e. a function from \( \omega \) to \( S \)) such that \((s_{i-1}, a_i, s_i) \in \Delta \), for all \( i \geq 1 \). A run \( \sigma = s_0, s_1, \ldots \) 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 \( \omega \)-word \( w \) is accepted by \( A \) if there is an accepting run of \( A \) over \( w \). The set of \( \omega \)-words accepted by \( A \) is denoted \( L_\omega(A) \).

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 \( A_1 = (\Sigma_1, S_1, \Delta_1, s_{01}, F_1) \) and \( A_2 = (\Sigma_2, S_2, \Delta_2, s_{02}, F_2) \) is the automaton \( A = (\Sigma, S, \Delta, s_0, \mathcal{F}) \) defined by

\[
\begin{align*}
\Sigma &= \Sigma_1 \cup \Sigma_2, \\
S &= S_1 \times S_2, \; s_0 = (s_{01}, s_{02}), \\
\mathcal{F} &= \bigcup_{F_j \in F_1} \{ F_j \times S_2 \} \cup \bigcup_{F_j \in F_2} \{ S_1 \times F_j \} \\
((s, t), a, (u, v)) &\in \Delta \text{ when} \\
- a &\in \Sigma_1 \cap \Sigma_2 \text{ and } (s, a, u) \in \Delta_1 \text{ and } (t, a, v) \in \Delta_2, \\
- a &\in \Sigma_1 \setminus \Sigma_2 \text{ and } (s, a, u) \in \Delta_1 \text{ and } v = t, \\
- a &\in \Sigma_2 \setminus \Sigma_1 \text{ and } u = s \text{ and } (t, a, v) \in \Delta_2.
\end{align*}
\]

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 \( s \in F_i \) for all \( F_j \in \mathcal{F} \), 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 \( \varphi \), it is possible to build a generalized Büchi automaton \( A_\varphi \) that accepts exactly the infinite words satisfying the temporal formula \( \varphi \) (the alphabet of this automaton is \( 2^P \) where \( P \) is the set of propositions appearing in the formula \( \varphi \)) [48, 44, 46]. 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 \( \varphi \) (one uses the negation of the formula as this yields a more efficient algorithm). The resulting automaton is \( A_{\neg \varphi} \).
2. Compute the product \( A_G = \prod_{1 \leq i \leq n} A_i \times A_{\neg \varphi} \) (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 \( F_i \) of its set \( \mathcal{F} \) of accepting sets. This can be done with a linear-time algorithm [1]. 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 [30] 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 \( \Sigma \), and where \( D \) is a binary, symmetrical, and reflexive relation on \( A \) called the dependency in \( \Sigma \).

\[ I_\Sigma = A^2 \setminus D \text{ stands for the independency in } \Sigma. \]

**Definition 3.2** Let \( \Sigma \) 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 \( \varepsilon \) 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 \( \Sigma \).

**Definition 3.3** Equivalence classes of \( \equiv_\Sigma \) are called traces over \( \Sigma \).

A trace characterized by a word \( w \) and a concurrent alphabet \( \Sigma \) 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 \[ab\] 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 \(A_{-f}\). From now on, \(A_{-f}\) 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 \(t = (s, a, s') \in \Delta\) with \(s = (s_1, s_2, \ldots, s_{n+1})\) and \(s' = (s'_1, s'_2, \ldots, s'_{n+1})\), the sets (by extension, we consider the states of \(A_G\) as sets in the following definitions)

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

are called respectively the preset, the postset and the proximity of the transition \(t\). Intuitively, the preset, resp. the postset, of a transition \(t = (s, a, s')\) 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 \(t_1 = (s_1, a_1, s'_1), t_2 = (s_2, a_2, s'_2) \in \Delta\) are said to be equivalent (notation \(\equiv\)) iff

\[
\bullet t_1 = \bullet t_2 \land \bullet t_1^* = \bullet t_2^* \land a_1 = a_2.
\]

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 this transition. We denote by \(T\) the set of equivalence classes defined over \(\Delta\) by \(\equiv\).

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

\[
(t_1, t_2) \in D_{A_G} \iff \bullet t_1^* \cap \bullet t_2^* \neq \emptyset.
\]

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).

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 accepted by \(A_G\) (all states of \(A_G\) considered accepting). 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 \(A_G\) [22].

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

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

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

In [22] an algorithm for constructing a trace automaton corresponding to a concurrent program 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 algorithm presented in [22] 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 [22]) 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.

\footnote{In [22] a concurrent program is represented by a contact-free one-safe F/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 [22] is still applicable in the context considered here.}
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 $A_1, 1 \leq i \leq n$, 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 $F$. 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 $F_T = \{F'_1, \ldots, F'_k\}$ be the set of sets of accepting states of $A_T$. The set $F_T$ of sets of accepting states of $A_T$ is then defined by $F_T = \{F'_1, \ldots, F'_k\}$ with $F'_i = F_i \cap S_T$.

Even if we extend the definition of $A_T$ to include the set $F_T$ defined above (let us call the result $A_T^\infty$), we still cannot use $A_T^\infty$ 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 $A_T^\infty$ 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 $A_T^\infty$ 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 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 $A_T^\infty$ 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 $F = \{\{s_1\}, \{s_2\}\}$ and $F' = \{\{s'_1\}, \{s'_2\}\}$ respectively. A possible trace automaton $A_T^\infty$ is given in Figure 3. Its set of sets of accepting states is defined by $F_T = \{\{(s_1, s'_0), (s_1, s'_1), (s_1, s'_2)\}, \{(s'_2, s_0)\}, \{(s_1, s'_1)\}, \{(s_1, s'_2)\}\}$. 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 $A$](image)

![Figure 2: Generalized Büchi automaton $A'$](image)

We now formalize the above discussion. Let $A_G$ and $A_T^\infty$ 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 $A_T^\infty$ on an infinite word $w$. One can view this computation as an infinite sequence of transitions each of which is an element $(s, a, s')$ of $S \times \Sigma \times S$. For any transition of $A_G$ or $A_T^\infty$, 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 $A_T^\infty$ 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 $\kappa$ of $A$ to the automaton $A_i$ (denoted $\kappa|A_i$) is the subsequence of $\kappa$ that contains only the transitions for which $A_i$ is active.

Note that the restriction of a computation of $A_G$ or $A_T^\infty$ to an automaton $A_i$ is in fact a computation of $A_i$. However, the restriction can be finite, even if the initial computation is infinite. We can nevertheless prove the following.

**Theorem 4.1** Let $\kappa$ be a computation (finite or $\omega$-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 $\kappa_i$ (finite or $\omega$-infinite) of the trace automaton $A_T^\infty$ such that $\kappa|A_i = \kappa_i|A_i$.

Note that it is not true that there is a single computation $\kappa'$ of $A_T^\infty$ such that $\kappa'|A_i = \kappa_i|A_i$ 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 $F$. We can prove the following.

**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 $A_T^\infty$ be the product and trace automaton 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 $A_T^\infty$ is nonempty.

Theorem 4.2 is obtained from Theorem 4.1 and from the immediate fact that all computations of the trace automaton are also computations of the product 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. 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 $F$.

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

Trace automata do not adequately represent the $\omega$-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 $\omega$. 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 [9, 8]).

A word of length $\omega \times n$ over the alphabet $\Sigma$ is a function $w$ from the ordinal $\omega \times n$ to $\Sigma$. We use automata that are defined exactly as in Section 2 and simply change the definition of a run. A run of an automaton $A = (\Sigma, S, \Delta, s_0, \mathcal{F})$ on a word $w$ of length $\omega \times n$ is a function $\sigma$ 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 $\alpha$ whose limit is $\lambda$ such that $\sigma(\alpha) = \sigma(\lambda)$. 

Figure 3: Trace automaton $A_T^\infty$
The notions of accepting run and accepted word are essentially unchanged. A run $\sigma$ 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 run of $A$ over $w$. The set of $\omega \times n$ words accepted by $A$ is denoted $L_{\omega \times n}(A)$.

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

**Theorem 5.1** Let $A = (\Sigma, S, \Delta, s_0, \mathcal{F})$ be an automaton. Then, $L_{\omega \times n}(A) \neq \emptyset$ if and only if there is a sequence of 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 $F_j \cap C_i \neq \emptyset$.

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

**Theorem 5.2** Let $\kappa$ be an $\omega$-computation of the global automaton $A_G$ obtained by composing the automata $A_i$, $1 \leq i \leq n+1$. Then, there is a computation $\kappa'$ of length at most $\omega \times (n+1)$ of the trace automaton $A_T^{\omega \times n}$ such that for all $1 \leq i \leq n+1$, $\kappa|_{A_i} = \kappa'|_{A_i}$.

To use the trace automaton for model checking, we also need the converse of Theorem 5.2. 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 $\omega$. For this to be possible we need to put some restrictions on computations. Consider a computation of length $\omega \times (n+1)$. For each $\omega$-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 $\omega$-sequence as its suffix that only contains states that appear infinitely often. The rest of the $\omega$-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 5.2 holds for separable computations and hence the following.

**Theorem 5.3** Let $A_i$, $1 \leq i \leq n+1$ be generalized Büchi automata. Let $A_G$ and $A_T^{\omega \times n}$ be the product and trace automata obtained by composing the automata $A_i$. Then, the automaton $A_G$ is nonempty (has at least one accepting computation) iff the trace automaton $A_T^{\omega \times n}$ has at least one separable computation of length at most $\omega \times (n+1)$.

Furthermore, note that Theorem 5.1 can be adapted to separable computations by requiring 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 $C_j$. Combining this observation with Theorem 5.3 we have a criterion for solving the model-checking problem in terms of the trace automaton $A_T^{\omega \times n}$.

6 Conclusions and Comparison with Other Work

The closest work to the one presented here is certainly that of Valmari [40]. However, while it has the same goal it does not achieve the same results. Indeed, Valmari only handles a temporal logic where the operator “next” cannot appear. We handle the full logic and, actually, we can also handle extended temporal logics like that of [45]. Also, in [40] concurrency is modelled by interleaving for all actions that appear in the formula. Because in our method the automaton for the formula is combined with the program, we do not have this drawback. Moreover, to solve the problem that trace automata do not adequately represent the infinite behaviors of a set of processes, Valmari has to modify the construction of the automaton and actually build an automaton that will usually have more states and transitions. We solve this problem by viewing the trace automaton as an automaton on $\omega \times n$-words. Finally, the possibility of modelling fairness conditions within the program is not present in [40]. All the advantages above are linked to the strategy we use for solving the model-checking problem which, as we have shown, is quite distinct from Valmari’s. However, at the heart of both methods lies an algorithm for computing an automaton that only represents “some” interleavings of concurrent events: Valmari uses an algorithm based on “Stubborn Sets”, we use the construction of the “Trace Automaton” given in [22]. This algorithm also influences the effectiveness of a model-checking method. Unfortunately, this influence is not at all as clear cut as that of the strategy. It is quite possible that for some problems the “Trace Automaton” algorithm is best whereas for others the “Stubborn Sets” one is preferable.

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 [10] that we can check systems with astronomical numbers of (interleaving semantics) states. Of course this is rather meaningless since the trick is not to explore all states either by treating classes of states as one state (the approach of [10]) or by completely avoiding parts of the state space (our approach). The only real fact we can give is that experimental results with trace automata are very encouraging.

Finally, note that our method has the advantages of “on the fly verification” [16, 26, 4, 24]. 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 David Dill, Antti Valmari and anonymous referees for helpful comments on this paper.

References


