Appears in: CONCUR '93 Proceedings, Lecture Notes in Computer Science, vol. 715, Hildesheim, August, 1993, Springer-Verlag, pp. 233--246.

Partial-Order Methods for Temporal Verification*

Pierre Wolper and Patrice Godefroid

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

Abstract. A common approach for verifying a concurrent system is to compute the product of finite-state descriptions of the processes involved. Unfortunately, the size of this product is frequently prohibitive due, among other causes, to the modelling of concurrency by interleaving. However, computing all interleavings of concurrent events is not a priori necessary for verification: interleavings corresponding to the same concurrent execution contain related information. One can thus hope to be able to verify properties of a concurrent system without computing all interleavings of its executions. This paper overviews a collection of techniques that make this possible.

1 Introduction

Verification methods for concurrent systems come in enough flavors to make the task of the systematic sampler a lifelong quest for his preferred choice. Of course, there are good reasons for this proliferation: there is no single concurrency paradigm and the necessary variety of semantical models naturally leads to a palette of verification approaches. Nevertheless, there is a central ingredient to essentially all concurrent system verification schemes: a method for reasoning about the concurrent combination of executions. This method is usually either operational or logical. Operational methods use expansion theorems that make it possible to compute a nondeterministic representation of the concurrent execution of processes. Logical methods rely on assertions for describing the components to be combined, and use a composition rule to derive therefrom an assertion about the joint behavior of these components. Applying such compositional rules generally requires the ability to prove nontrivial theorems in an underlying logical system.

The research described in this paper starts from the point of view that a very crucial factor in the applicability of a proof method for concurrent systems is the ease of use of its parallel composition scheme. Indeed, this is the central step of most verifications and it is far from sufficient just to know that it can be applied

* This work was supported by the Esprit BRA action REACT (6021) and by the Belgian Incentive Program "Information Technology" - Computer Science of the future, initiated by the Belgian State - Prime Minister's Office - Science Policy Office. The scientific responsibility is assumed by its authors.

(e.g., because the proof system is complete): the real problem is to apply it! We thus consider that this step must be either algorithmic or within the reach of a theorem prover helped by moderate human interaction. Our choice is to focus on algorithmic approaches. Obviously, for these, an essential characteristic is their complexity and, beyond worst-case complexity, their behavior on typical systems. We will thus be concerned by the development of efficient algorithms for composing concurrent processes in the context of verification and present an overview of some recent results obtained in this area.

We work in the standard and simple context of concurrent systems modelled by communicating finite automata. The verification problem we consider is to check that the execution sequences produced by such a system fall within a given set specified by, for instance, a linear-time temporal logic formula [Wol89]. In this context, the traditional concurrent composition mechanism that is used for verification is to compute the product of the finite automata. The goal is to improve on this straightforward approach.

The intuition behind the methods we discuss is that concurrent executions are really partial orders and that expanding such a partial order into the set of *all* its interleavings is an inefficient way of analyzing concurrent executions. However, rather than choosing to work with direct representations of partial-orders, the methods described here keep to an interleaving representation of partial-orders, but attempt to limit the expansion to just *some* representative interleavings of each partial-order computation. Hence the name "partial-order methods". It must nevertheless be noted that, though they are very inspired by partial-order semantics (especially by Mazurkiewicz's traces [Maz86]), these methods do not comply with any specific partial-order semantics. Indeed, the only requirement is that the modified concurrent composition computes enough interleavings to make checking the desired property possible. Not all concurrent executions need be represented if the verification does not require it and, conversely, a given concurrent execution can be represented by several redundant interleavings. The prime concern is to check the desired property as efficiently as possible.

In the following section we describe the basic model and the concepts that will be used throughout the paper. Next, we introduce the two families of algorithms that have been used in the class of methods we discuss: persistent sets [Ove81, Val90, Val91, GW91b] and sleep sets [God90, GW93]. We describe these algorithms and prove their correctness in the simple case where the properties of interest are simply termination properties. Thereafter, we discuss how these algorithms can be extended for handling more general properties ranging from safety properties to general temporal properties. Finally, the paper ends with a section evaluating the effectiveness of the methods we describe and relating them to other families of "partial-order" approaches.

2 Basic Notions

We consider a concurrent program P composed of n concurrent processes P_i. Each process is described by a transition system $A_i = (\Sigma_i, S_i, \Delta_i, s_{0i})$, where Σ_i

is an action alphabet, S_i is a finite set of states, $\Delta_i \subseteq S_i \times \Sigma_i \times S_i$ is a transition relation, and $s_{0i} \in S_i$ is the starting state.

The concurrent behavior of these processes is defined by the usual operational semantics (CSP-style): actions that appear in several processes are synchronized, others are interleaved. Using this convention, one can obtain a *global* transition system A_G describing the joint behavior of the processes P_i. This global transition system is traditionally computed by constructing the reachable states of the product of the transition systems describing the n processes P_i. Precisely, this product is the transition system $A = (\Sigma, S, \Delta, s_0)$ where

- $\Sigma = \bigcup_{1 \leq i \leq n} \Sigma_i$,
- $S = \prod_{1 \leq i \leq n} S_i$ (Cartesian product), we denote the ith component of a state $s \in S$ by $s(i)$,
- $s_0 = (s_{01}, \ldots, s_{0n})$,
- $(s, a, s') \in \Delta$ if and only if
  - for all $1 \leq i \leq n$ such that $a \in \Sigma_i$, $(s(i), a, s'(i)) \in \Delta_i$, and
  - for all $1 \leq i \leq n$ such that $a \notin \Sigma_i$, $s(i) = s'(i)$.

The transition system A_G is then A restricted to its states reachable from s_0.

Our goal is to be able to check properties of A_G without computing the whole of A_G. The motivation is obvious : A_G is often much too large to be computed! Note that the definition of the problem we address relies exclusively on traditional interleaving semantics. Also note that for clarity we have chosen a particular model of process communication (action synchronization), but that this choice is not essential. We could also consider processes communicating through shared objects such as variables or channels. What matters is that it is possible to compute a global transition system representing the joint behavior of the processes.

The basic idea that enables us not to construct the whole of A_G is the following: A_G contains many paths that correspond simply to different execution orders of the same program steps. If these steps are "independent", for instance because they are executed by noninteracting processes, then changing their order will not modify their combined effect. Thus, it is not necessary to explore all execution orders (interleavings) of independent program steps. To go further, we need to make precise the notions of program step and independent program steps.

Intuitively, two *global* transitions t_1 and t_2 of A_G correspond to the same program step (*program* transition) if they correspond to the same set of process transitions. Formally, one can view a program transition as an equivalence class of global transitions. In the context defined above, two global transitions $t_1 = (s_1, a_1, {s'}_1)$ and $t_2 = (s_2, a_2, {s'}_2)$ correspond to the same program transition if and only if

- $a_1 = a_2 = a$ (the action is the same) and,
- for all $1 \leq i \leq n$ such that $a \in \Sigma_i$, $s_1(i) = s_2(i)$ and ${s'}_1(i) = {s'}_2(i)$ (the state changes for the processes participating in the transition are identical).

In what follows, we will only talk about *program* transitions which will be simply called transitions. A *global* transition will be identified with the program transition to which it belongs. The set of transitions of a program P will be denoted by T. Finally, we will say that a program transition t is *enabled* in a state s of A_G if some global transition in the equivalence class defining t originates in s.

When should we consider transitions to be independent? Our intuitive idea is that transitions are independent if their order does not matter. We use the following definition (adapted from [KP92]).

Definition 1. Two (program) transitions t_1 and t_2 of a program P with a global transition system A_G are *independent* if the following two conditions are true in all states s of A_G (otherwise they are said to be *dependent*):

1. if t_1 (t_2) is enabled in s and[2] $s \xrightarrow{t_1} s'$ $(s \xrightarrow{t_2} s')$, then t_2 (t_1) is enabled in s iff t_2 (t_1) is enabled in s' (independent transitions can neither disable nor enable each other); and
2. if t_1 and t_2 are enabled in s, then there is a unique state s' such that both $s \stackrel{t_1 t_2}{\Rightarrow} s'$ and $s \stackrel{t_2 t_1}{\Rightarrow} s'$ (commutativity of enabled independent transitions).

One can wonder if this definition is of more than semantic use. In fact, in most programming models, it is easy to give easily checkable syntactic conditions that are sufficient for transitions to be independent. For instance in the model introduced above, two transitions t_1 and t_2 are independent if the set of processes that participate in t_1 (i.e. such that the action of t_1 is in their alphabet) is disjoint from the set of processes that participate in t_2.

Following the work of Mazurkiewicz [Maz86], one can use the notion of independent transitions to define an equivalence relation on sequences of transitions: two sequences of transitions are equivalent if they can be obtained from each other by successively permuting adjacent independent transitions. Thus, given an independency relation, sequences of transitions can be grouped into equivalence classes which Mazurkiewicz calls traces. The trace containing a sequence of transitions w will be denoted $[w]$. It is easy to see that sequences of transitions w_1 and w_2 belonging to the same trace lead to the same state of A_G:

Property 2. If $s \stackrel{w_1}{\Rightarrow} s_1$, $s \stackrel{w_2}{\Rightarrow} s_2$ and $[w_1] = [w_2]$, then $s_1 = s_2$.

This property is basically what will allow us to only explore part of the global state-space A_G: to determine if a state is reachable by a trace, it is sufficient to explore *one* transition sequence corresponding to that trace.

It might thus appear that we are using Mazurkiewicz's trace semantics. This is not really so. Indeed, to view Mazurkiewicz's theory as a semantics, the independency relation should be considered as part of the semantics: given an independency relation, one can determine the Mazurkiewicz semantics of a program. The criterion for a partial construction of the state-space would then be

[2] We write $s \xrightarrow{t} s'$ to mean that the transition t leads from the state s to the state s' and $s \stackrel{w}{\Rightarrow} s'$ to mean that the sequence of transitions w leads from s to s'.

that the Mazurkiewicz semantics are preserved. Here a less restrictive point of view is taken. The semantic criterion is that the result of checking a property in the class of interest should be the same as if checking the property on A_G. The link with Mazurkiewicz semantics is only in the fact that the algorithms presented in the next section rely on the concept of independency and on the properties it implies, especially Property 2. The algorithms can actually use any independence criterion that implies the one of Definition 1. In some algorithms it is even possible to use notions of independence that are weaker than the one of Definition 1 [GP93, Val91].

3 The Algorithms

In this section, we present the basic algorithmic ideas used in the style of partial-order verification methods we are describing. For simplicity we only consider the problem of detecting terminating (or deadlock) states. The verification of more general properties will be discussed in the following section. The specification of the algorithms we present is thus that they should find all states of A_G with no outgoing transitions while exploring as small a fraction as possible of A_G.

All algorithms we will present follow the same basic pattern: they operate as classical state-space searches except that, at each state s reached during the search, they compute a subset T of the set of transitions enabled at s and explore only the transitions in T, the other enabled transitions are not explored. We call such a search a *selective search*. It is easy to see that a selective search through A_G only reaches a subset (not necessarily proper) of the states and transitions of A_G.

Two main techniques for computing such sets T have been proposed in the literature. The first technique is actually a whole family of algorithms [Ove81, Val91, GW91b] that compute what we call "persistent sets". The most elaborate technique of this family that has been proposed so far is the stubborn set technique of Valmari [Val91]. The second type of technique is the sleep set technique of Godefroid and Wolper (e.g., see [GW93]). Interestingly, these two techniques are perfectly compatible and can be used simultaneously to further improve the selection of the set T. We first describe "persistent set" techniques.

3.1 Persistent Sets

Intuitively, in a state s of A_G, a subset T of the transitions enabled in s is called *persistent* if all transitions not in T that are executable from s, or from a state reachable from s through transitions not in T, are independent with respect to all transitions in T. In other words, whatever one does from s, while remaining outside of T, does not interact with or affect T. Formally, we have the following.

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

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

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

Note that the set of all enabled transitions in a state s is trivially persistent since nothing is reachable from s by transitions that are not in this set.

Let a *persistent-set selective search* be a selective search of A_G which, at each state s that it reaches, selects a nonempty set of enabled transitions T that is persistent in s. We now prove that such a search reaches all deadlock (termination) states of A_G.

Theorem 4. *Let s be a state reached in a persistent-set selective search, and let d be a deadlock state. If $\exists w : s \stackrel{w}{\Rightarrow} d$, then d will also be reached by the persistent-set selective search.*

Proof. The proof proceeds by induction on the length of w. For $|w| = 0$, the result is immediate. Now, assume the theorem holds for a path (sequence of transitions) w of length $n \geq 0$ and let us prove that it holds for a path w' of length $n + 1$.

Let the path w' be $t_0, t_1, \ldots, t_n$ and let $s = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} d$ be the sequence of states it goes through. Now, let **persistent**(s) be a nonempty persistent set in state s. Assume first that $t_0, t_1, \ldots, t_n \notin$ **persistent**(s). Then by the definition of persistent sets, all transitions t_i, $0 \leq i \leq n$, are independent with all transitions of **persistent**(s). Thus all transitions in **persistent**(s) are still enabled in d which can thus not be a deadlock. Hence, some transition of the path leading to d must be in **persistent**(s).

Let thus t_i be the first transition in w' that is in **persistent**(s) and let w'' be the sequence $t_i t_0 t_1 \ldots t_{i-1} t_{i+1} \ldots t_n$, i.e. the sequence w' where the transition t_i has been moved to the first position. By definition of persistent sets, we have that $\forall 0 \leq j < i$: t_j is independent with respect to t_i. Consequently, $[w'] = [w'']$ (they belong to the same trace) and since $s \stackrel{w'}{\Rightarrow} d$, by Property 2 we also have that $s \stackrel{w''}{\Rightarrow} d$. Since t_i is in **persistent**(s), it is explored from state s and a state from which a path of length n leads to the deadlock d is reached by the search. This together with the inductive hypothesis proves the theorem.

From Theorem 4 it is then immediate to conclude that a persistent-set selective search started from the initial state of A_G will explore all deadlocks of A_G.

Of course, the key element required for the implementation of a persistent-set selective search is an algorithm for computing persistent sets. All algorithms that have been proposed for this purpose [Ove81, Val91, GW91b] infer the persistent sets from the static structure (code) of the program being verified. They differ by the type of information about the program that they use (e.g., "distinguishing between local and global transitions", "which process can access which variable", "which process can access which variable from its current location", etc.). The aim of these algorithms is to obtain the smallest possible persistent sets. Usually, the more information about the program the algorithm uses, the smallest the persistent set it produces are, albeit at the cost of a higher computational complexity. See [HGP92] for a brief comparison of several of these algorithms. Note that exploring the smallest number of enabled transitions at each step of the search is only a heuristic: it does not necessary lead to the exploration of the smallest number of states.

The most elaborated technique for computing persistent sets is the stubborn set technique of Valmari. Stubborn sets[3] can be formally defined as follows (adapted from [Val91]):

Definition 5. A set T_s of transitions is a stubborn set in a state s if T_s contains at least one enabled transition and $\forall t \in T_s$ the following two conditions hold:

1. if t is disabled in s, and c_i is a necessary condition for t to be enabled which is false in s, then all transitions t' whose execution can make c_i *true* are also in T_s;
2. if t is enabled in s, then all transitions t' such that t and t' are dependent are also in T_s.

It can be shown that, by taking all enabled transitions of a stubborn set, one obtains a persistent set. From the above definition, one can derive algorithms for computing stubborn sets [Val91].

3.2 Sleep Sets

The second technique for computing the set of transitions T to consider in a selective search is the sleep set technique [GW93]. This technique does not exploit information about the static structure (code) of the program, but rather about the past of the search. Used alone it reduces the number of transitions explored, but not the number of states, which can still be very useful [GHP92]. Used in conjunction with a persistent set technique it can further reduce the number of states explored. Indeed, when the persistent set technique cannot avoid the selection of *independent* transitions in a state, sleep sets can avoid the exploration of multiple interleavings of these transitions.

Let us consider an example to illustrate the basic idea behind sleep sets. Consider a state s reached during the search and assume two enabled independent transitions t_1 and t_2 are selected (for instance, by a persistent set algorithm) to be explored from s. After exploring t_1, t_2 is still enabled since t_1 and t_2 are independent, and t_2 could thus be selected to be explored from the state s_1 reached by executing t_1 from s. Conversely, after exploring t_2, t_1 will, for the same reason, also still be enabled and could also be selected to be explored from the state s_2 reached by executing t_2 from s. When selecting two independent transitions t_1 and t_2 from s, there is thus a risk that the two interleavings of t_1 and t_2, i.e., the two sequences t_1t_2 and t_2t_1, will be explored. This is potentially wasteful since both of these interleavings lead to the same state. In order to prevent this

[3] "Stubborn sets in the strong sense" according to Valmari's terminology. "Stubborn sets in the weak sense" will not be considered here.

situation from occurring, the sleep set method prevents the exploration of t_1 in s_2.

More precisely, a *sleep set* is a set of transitions. A sleep set is computed for each state s reached during the search. The sleep set computed for a state s is a set of transitions that are *enabled* in s but *will not be executed* from s. The sleep set associated with the initial state s_0 is the empty set. The sleep sets of the successors of a state s are then computed as follows.

Let T be the set of transitions that have been selected to be explored from s. Take a first transition t_1 out of T. The sleep set associated with the state reached by t_1 is the sleep set of s unmodified except for the elimination of transitions that are dependent with t_1. Let t_2 be the second transition taken out of T. The sleep set associated with the state reached by t_2 from s is the sleep set of s augmented with t_1 minus all transitions that are dependent with t_2. One proceeds in a similar way with the remaining transitions of T. The general rule is thus that the sleep set of a state s' reached by a transition t from a state s is the sleep set that was obtained when reaching s augmented with all transitions already taken from T and purged of all transitions that are dependent with t.

```
Initialize:Stack is empty; H is empty;
          s0.Sleep = ∅;
          push (s0) onto Stack;
Loop: while Stack ≠ ∅ do {
          pop (s) from Stack;
          if s is NOT already in H then {
            T = Persistent_Set(s) \ s.Sleep;
            if T = ∅ ∧ s.Sleep = ∅ then print "Deadlock!";
            enter s in H
            }
          else {
            T = {t | t ∈ H(s).Sleep ∧ t ∉ s.Sleep};
            s.Sleep = s.Sleep ∩ H(s).Sleep;
            H(s).Sleep = s.Sleep
            }
          for all t in T do {
              s' = succ(s) after t; /* execution of t */
              s'.Sleep = {t' | t' ∈ s.Sleep ∧ (t, t') are independent };
              push (s') onto Stack;
              s.Sleep = s.Sleep ∪ {t}
              }
          }
```

Fig. 1. Deadlock Detection Algorithm

The algorithm of Figure 1 represents a persistent-set selective search augmented with all operations required to manipulate sleep sets. It uses a *Stack* and a hash table H used to store visited states and their sleep set. Each time a new state s is encountered during the search, a call to the function **Persistent_Set** is performed. This function returns a nonempty persistent set of transitions T, if such a set exists. Transitions that are in the current sleep set $s.Sleep$ need not be explored, and are thus removed from set T (line 7). If there is no enabled transition, a deadlock is reported (line 8). If the current state s has already been visited with a sleep set $H(s).Sleep$ containing transitions that are not in the current sleep set $s.Sleep$, these transitions are explored (line 12) with a new value of $Sleep$ equals to $s.Sleep \cap H(s).Sleep$ (lines 13–14). Once the transitions that have to be explored are determined, the procedure described above is used to compute the sleep set that is to be associated with each successor of s (lines 16–21).

The correctness proof of the algorithm is the following.

Theorem 6. *Let d be a deadlock reachable from the initial state s_0. Then, there is a path from s_0 leading to d which is explored by the algorithm of Figure 1.*

Proof. Consider the algorithm of Figure 1. Let s be a state visited by this algorithm, $s.Sleep$ be the sleep set associated with this state and stored with it in *Stack*, and d be a deadlock. We first prove that, if $\exists w : s \overset{w}{\Rightarrow} d$, and if, for all $w_i \in [w]$, the first transition of w_i is not in $s.Sleep$, then, if s is pushed onto *Stack* (i.e., is visited), d is visited.

The proof proceeds by induction on the length of w. For $|w| = 0$, the result is immediate. Now, assume the proposition holds for paths of length $n \geq 0$ and let us prove that it holds for a path w of length $n+1$. Let s be a visited state such that $\exists w : s \overset{w}{\Rightarrow} d$ and $\forall w_i \in [w] : t_i \notin s.Sleep$, where t_i denotes the first transition of the path w_i.

We first consider the case where s is not already in H. In this case, we know from the proof of Theorem 4 that at least one of the transitions t_i is selected by the function **Persistent_Set**(s). Moreover, this transition is executed since it is not in $s.Sleep$. If there are several such transitions, we consider the first of these, which we will call t_1. Let s' be the successor of s reached by executing t_1. We have $w_1 = t_1 w'$. By Property 2, since w leads to d in $n+1$ steps, w_1 also leads to d in $n+1$ steps and, consequently, w' also leads to d from s' and is of length n. Moreover, we can also show that for all $w'_i \in [w']$, the first transition of w'_i is not in $s'.Sleep$.

Indeed, assume the opposite, i.e., there exists some transition $t'_i \in s'.Sleep$ such that t'_i is the first transition of a $w'_i \in [w']$ leading to d. This implies that t'_i and t_1 (the transition leading to s') are independent, else t'_i would not have been passed on to the sleep set associated to s'. Since they are independent, t'_i is enabled in s and is the first transition of a path w_i leading from s to d. Given that t'_i is in $s'.Sleep$, either t'_i was in the sleep set $s.Sleep$ or was added after being executed from s. The first possibility is in contradiction with the fact that t'_i is also the first transition of some $w_i \in [w]$ leading to d from s and thus is assumed not to be in $s.Sleep$. The second possibility is incompatible with the fact that t_1 is the first such transition to be executed from s. The inductive hypothesis can thus be used to establish that d is visited from s' and thus from s.

We now consider the case where the state s already appears in the table H. Let $H(s).Sleep$ be the sleep set stored with s in H (while $s.Sleep$ represents the sleep set stored with s on the $Stack$). There are two situations: either some of the t_i transitions are in $H(s).Sleep$, or none of them are. In the first situation, all the transitions in $H(s).Sleep$ and not in $s.Sleep$ are executed. Thus, at least some t_i transitions will be executed and, by an argument identical to the one above, we can then establish that the first among these leads to a state s' in which our inductive hypothesis can be applied.

Let us now turn to the second situation in which none of the t_i transitions are in $H(s).Sleep$. This can be the case either because no t_i transition was in the sleep set the very first time s was introduced in H or because the t_i transitions were all removed when s was reached again later in the search. Under the first hypothesis, a state s' in which the inductive hypothesis can be applied was thus visited the first time transitions from s were explored. Under the second hypothesis, consider the last visit of s where some of the t_i transitions were executed from s, and let t be the first transition among those; t leads to a successor state s' in which the inductive hypothesis can be applied.

We have just proved that, if $\exists w : s \stackrel{w}{\Rightarrow} d$, and if, for all $w_i \in [w]$, the first transition of w_i is not in $s.Sleep$, then d is visited from s. By applying this result to the initial state s_0, we directly reach the conclusion that the algorithm of Figure 1 indeed reaches all deadlock states, since the sleep set associated to the initial state is the empty set.

4 Checking General Properties

In order to check for properties more elaborate than deadlocks, it is usually necessary to preserve more information in the reduced state space. Thus the selective search algorithm has to be adapted to the type of property one wants to check. We briefly discuss how this can be done for several common classes of properties.

The first class of properties we consider is reachability of a *local* state ℓ of a process P_i. This amounts to checking for the reachability of *some* global state s such that $s(i) = \ell$.

The algorithm presented in the previous section is not sufficient for checking such a property. This is illustrated by the following example. Consider a program composed by two processes P_1 and P_2. P_1 consists of one looping transition $t_1 = (s_{01}, a, s_{01})$, while P_2 can perform a transition $t_2 = (s_{02}, b, \ell)$ from its initial state s_{02} to its final state ℓ. If one executes the algorithm of Figure 1, in the initial state (s_{01}, s_{02}) of the system, the algorithm can, for instance, select a persistent set containing just t_1. After exploring t_1, the selective search stops since transition t_1 leads back to the initial state (s_{01}, s_{02}) which has already been visited with an empty sleep set. Transition t_2, though being enabled, has never been explored, and local state ℓ has not been reached. This is sufficient for proving the absence of deadlock: since P_1 can loop forever independently of the rest of the system, i.e., process P_2, one can conclude that this program is deadlock free even without considering the possible behaviors of P_2. However, this is not sufficient to determine if ℓ is reachable or not from the initial state.

The problem illustrated above is sometimes referred to as the "ignoring problem" [Val91]: the behavior of some processes (e.g., P_2 in the above example) can be completely ignored during a selective search. Various solutions have been proposed to solve this problem (e.g., [Val91, HGP92]). The idea is to enforce additional conditions during the search that ensures that the choices between enabled independent transitions is not completely "unfair" with respect to some processes. It can be shown that the reachability of a local state can be checked by such a "fair" selective search.

Many interesting properties can be reduced to local state reachability. The general idea is that if a property is not local to a process, one introduces an additional process in the system to which it is local. For instance, checking an invariant, i.e., if a given condition remains true in all states of A_G, can be done by adding a process with a single transition $(s_0, test, s_{error})$ testing the truth value of the condition. In a similar way, user-specified assertions and dead code (i.e., program statements that are never executed) can be checked for. More generally, the verification of any safety property can be reduced to checking the reachability of a local state [AS87].

The verification of general temporal properties can also be reduced to a local property by introducing an additional process [VW86, Wol89, CVWY92] representing the formula to be checked. However, in this context one has to check not for reachable local states, but for cycles containing specific local states. This further strengthens the requirements on the selective search. This approach has been studied in [GW91a, Val93].

An alternative approach to checking temporal formulas with a selective search is to restrict the class of formulas that is considered. This must be done in such a way that it is still sound to check the formula on the reduced state space produced by the selective search. Of course, there is a trade off between the extent of the selective search and the class of properties that can be checked. The methods described in [Val90, Pel93] follow this paradigm.

5 Evaluation and Conclusions

How much can one gain by using the methods described in this paper? It is very difficult to give a general answer. Indeed, one can quite easily construct families of programs for which the growth of the state space is reduced from exponential to polynomial by a selective search; on the other hand, it is also easy to construct programs for which nothing is gained whatsoever. In this context, implementation and experiments with meaningful examples appears to be the most informative approach to evaluating the partial-order verification methods.

We have implemented the algorithms discussed in the previous sections in an add-on package for the protocol validation system SPIN [Hol91]. This Partial-Order Package is available free of charge for educational and research purposes by anonymous ftp from montefiore.ulg.ac.be (139.165.16.1) from the /pub/po-package directory.

This implementation has been tested on various real protocols[4]. The table below compares a classical exhaustive state-space search (Algorithm 1) to the results of our partial-order package (Algorithm 2). Two sample protocols are considered : URP and DTP.

- URP is AT&T's Universal Receiver Protocol, modeled by 419 lines of code. It consists of three processes communicating via FIFO channels.
- DTP is a data transfer protocol modeled by 391 lines of code. It consists of three processes communicating via FIFO channels.

Experiments were performed on a SPARC2 workstation (64 Megabytes of RAM). For each run, the numbers of visited states and traversed transitions are given. Time (in seconds) is user time plus system time as reported by the UNIX system time command.

Protocol	Algorithm	Stored States	Transitions	Time
URP	Algo 1	19,515	47,836	4.9
	Algo 2	4,430	4,659	7.9
DTP	Algo 1	251,409	648,467	59.8
	Algo 2	9,920	10,367	7.0

These results show that partial-order methods can yield substantial improvements in the performance of the verification. Sometimes the improvement is not very spectacular (URP), sometimes it is (DTP). Overall, our assessment is that the partial-order methods do provide substantial gains.

This paper has presented only one family of partial-order methods: those based on selectively constructing the interleavings of concurrent events. There are other approaches. For instance, in [McM92] one finds a verification method that works by unfolding a Petri net description of the concurrent system into a finite acyclic structure. Also, in [PL89, PL90] a method that relies on a pomset grammar description of the system is introduced.

References

[AS87] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[GHP92] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *Proc. 4th Workshop on Computer Aided Verification*, Montreal, June 1992.

[4] We thank Gerard J. Holzmann for providing us with these examples.

[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. Springer-Verlag.

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

[GW91a] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proc. 6th Symp. 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. Springer-Verlag.

[GW93] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, April 1993.

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

[Hol91] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International Editions, 1991.

[KP92] S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101:337–359, 1992.

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

[McM92] K. McMillan. Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. 4th Workshop on Computer Aided Verification*, Montreal, June 1992.

[Ove81] W. T. Overman. *Verification of Concurrent Systems: Function and Timing*. PhD thesis, University of California Los Angeles, 1981.

[Pel93] D. Peled. All from one, one for all: on model checking using representatives. In *Proc. 5th Workshop on Computer Aided Verification*, Elounda, June 1993. Lecture Notes in Computer Science, Springer-Verlag.

[PL89] D. K. Probst and H. F. Li. Abstract specification, composition and proof of correctness of delay-insensitive circuits and systems. Cs-vlsi-88-2, Department of Computer Science, Concordia University, Montreal, Quebec Canada, 1989.

[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. Springer-Verlag.

[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. Springer-Verlag.

[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. Springer-Verlag, 1991.

[Val93] A. Valmari. On-the-fly verification with stubborn sets. In *Proc. 5th Workshop on Computer Aided Verification*, Elounda, June 1993. Lecture Notes in Computer Science, Springer-Verlag.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. Symp. on Logic in Computer Science*, pages 322–331, Cambridge, June 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*, volume 398, pages 75–123. Lecture Notes in Computer Science, Springer-Verlag, 1989.

This article was processed using the LaTeX macro package with LLNCS style