Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties*

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 an algorithm for detecting deadlocks in concurrent finite-state systems without incurring most of the state explosion due to the modeling of concurrency by interleaving. For systems that have a high level of concurrency our algorithm can be much more efficient than the classical exploration of the whole state space. Finally, we show that our algorithm can also be used for verifying arbitrary safety properties.

1 Introduction

When reasoning assertionally about concurrent programs, separating safety and liveness properties is a well established paradigm [MP84, OL82]. Indeed, the proof techniques used for these two types of properties are quite different: safety properties are mostly established with invariance arguments whereas liveness properties require the use of well-founded orders.

Surprisingly, this distinction can often be ignored in model-checking [CES86, LP85, QS81, VW86]. Indeed, model checking can handle arbitrary temporal formulas which can represent both safety and liveness properties. Nevertheless, it has been noticed that restricting model-checking like techniques to safety properties can lead to better verification algorithms [BFG+91, BFH90, JJ89]. Intuitively, this can be understood by the fact that safety properties can be checked by only considering the finite behaviors of a system whereas liveness properties are only meaningful for infinite behaviors. Representing infinite behaviors requires the use of concepts such as automata on infinite words or trees [Büc62, Rab69] which are significantly harder to manipulate than their finite word or tree counterparts [Saf88, SVW87].

In this paper, we explore whether partial-order model-checking techniques such as those of [GW91, Val90] also benefit from being restricted to safety properties. The motivation for partial-order verification methods is that representing concurrency by interleaving is usually semantically adequate but quite often extremely wasteful. This wastefulness plagues model checking as well as other finite-state verification techniques since it creeps in when constructing the global state graph corresponding to a concurrent program. In [God90, PL90, Val89] among others, it is shown that most of the state explosion due to the modeling of concurrency by

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

interleaving can be avoided. For instance, the method used in [God90] is to build a state-graph where usually only one (rather than all) interleaving of the execution of concurrent events is represented. This reduced state-graph can then still be viewed as correctly representing the concurrent program by giving it an interpretation based on Mazurkiewicz's trace theory [Maz86].

We first turn to the grand-father of finite-state verification problems: deadlock detection. We show that for this problem, the algorithm of [God90] can be strongly simplified and improved. The main idea of the simplification is that since we are seeking deadlocks, one can generate a global representation of the program that chooses amongst independent events in a completely arbitrary way. The choices can even be completely "unfair" since, if there is a deadlock, favored processes will anyway eventually be blocked and the deadlock will be detected. This simplification leads to an algorithm that can be more easily and efficiently implemented than the one of [God90] and that often generates substantially fewer states.

Our next step is to turn to the verification of general safety properties. The approach we use here is that of "on the fly verification" [VW86, CVWY90, JJ89, HPOG89, FM91, JJ91]. Namely, we represent the safety property (or rather its complement) by an automaton on finite words and we check that the accepting states of this automaton are not accessible in its product with the automata representing the program. We thus reduce the problem of verifying safety properties to a state accessibility problem. This problem is in turn reduced by a simple transformation of the program and of the specification to a deadlock detection problem to which we apply our new algorithm.

The paper ends with a comparison between our contributions and related work.

2 A Representation of Concurrent Systems

We consider a concurrent program P composed of n concurrent processes P_i. Each process is described by a finite automaton A_i on finite words over an alphabet Σ_i. Formally, an automaton is a tuple $A = (\Sigma, S, \Delta, s_0)$, where Σ is an alphabet, S is a finite set of states, $\Delta \subseteq S \times \Sigma \times S$ is a transition relation, and $s_0 \in S$ is the starting state.

We consider automata without acceptance conditions. Thus a word $w = a_0 a_1 \dots a_{n-1}$ is accepted by an automaton A if there is a sequence of states $\sigma = s_0 \dots s_n$ such that s_0 is the starting state of A and, for all $0 \leq i \leq n-1$, $(s_i, a_i, s_{i+1}) \in \Delta$. We call such a sequence σ an *execution* of A on w. A state s is *reachable* from s_0 (notation $s_0 \stackrel{w}{\Rightarrow} s$) if there is some word $w = a_0 a_1 \dots a_{n-1}$ and some execution $\sigma = s_0 \dots s_n$ of A on w such that $s = s_n$.

An automaton A_G representing the joint global 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) automata $A_1 = (\Sigma_1, S_1, \Delta_1, s_{01})$ and $A_2 = (\Sigma_2, S_2, \Delta_2, s_{02})$ is the automaton $A = (\Sigma, S, \Delta, s_0)$ defined by

- $\Sigma = \Sigma_1 \cup \Sigma_2$,
- $S = S_1 \times S_2$, $s_0 = (s_{01}, s_{02})$,
- $((s,t), a, (u,v)) \in \Delta$ 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 $(t, a, v) \in \Delta_2$.

Let $\Delta \subseteq S \times \Sigma \times S$ denote the transition relation of the product A_G of the n automata A_i. For each transition $t = (\mathbf{s}, a, \mathbf{s}') \in \Delta$ with $\mathbf{s} = (s_1, s_2, \ldots, s_n)$ and $\mathbf{s}' = (s'_1, s'_2, \ldots, s'_n)$, the sets (by extension, we consider the states of A_G as sets in the following definitions)

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

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 $\equiv$.

3 Efficiently Detecting Deadlocks

A deadlock in a system composed of n concurrent processes P_i is defined as a reachable state of the system in which all processes P_i are blocked, i.e. where no transition is executable. Detecting deadlocks is usually performed by an exhaustive enumeration of all reachable states of the automaton A_G corresponding to the product of the n automata A_i. This enumeration amounts to exploring all possible transition sequences the system is able to perform and storing all intermediate states reached during this exploration. In concurrent systems, the number of reachable states can be very large: this is the well-known "state explosion" phenomenon. This combinatorial explosion limits both the applicability and the efficiency of the classical method.

In this section, we present a new simple method for detecting deadlocks without computing and storing all reachable states of the concurrent system. The basic idea is to describe the behavior of the system by means of partial orders rather than by sequences. More precisely, we use Mazurkiewicz's traces [Maz86] as a semantic model.

Traces are defined as equivalence classes of sequences. A trace represents a set of sequences defined over an alphabet Σ that only differ by the order of adjacent symbols which are independent according to a dependency relation D. For instance, if a and b are two symbols of Σ which are independent according to D, the trace $[ab]_{(\Sigma,D)}$ represents the two sequences ab and ba. A trace corresponds to a partial ordering of symbols and represents all linearizations of this partial order. If two independent symbols occur next to each other in a sequence of a trace, the order

```
1. Initialize: Stack is empty; H is empty;
               enter s0 in H;
               push (s0,∅) onto Stack;
2. Loop: while Stack ≠ ∅ do {
               pop (s,Sleep) from Stack;
               T = select_transitions(s,Sleep);
               if T = ∅ ∧ Sleep = ∅ then print "Deadlock!"
               for all t ∈ T do {
                   s' = (s \ •t) ∪ t•;
                   if s' is NOT already in H then {
                     enter s' in H;
                     push (s',sleep_attached_with(t)) onto Stack;
                     }
                   }
               }
```

Figure 1: Deadlock Detection Algorithm

of their occurrence is irrelevant since they occur concurrently in the partial order corresponding to that trace.

To describe the behavior of the concurrent system represented by A_G in terms of traces, we define the *dependency* in A_G as the relation $D_{A_G} \subseteq T \times T$ such that

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

The complement of D_{A_G} is called the *independency* in A_G. Given an alphabet and a dependency relation, a trace is fully characterized by only one of its linearizations (sequences). Thus, *given the set of transitions T and the dependency relation D_{A_G} defined above, the behavior of A_G is fully investigated by exploring only one sequence (interleaving) for each possible trace (partial ordering) the system is able to perform.*

Note that *all* deadlocks of A_G will be detected during this exploration. Indeed, for each deadlock, there exists at least one transition sequence w leading to this deadlock from the starting state. Moreover, all other sequences belonging to the same equivalence class $[w]_{(T,D_{A_G})}$ also lead to the same deadlock. Thus, for detecting this deadlock, it is sufficient to explore only one of these sequences which can be chosen arbitrarily. This deadlock-preserving property of partial-order semantics was already pointed out in [Gai88, Val88] among others.

Figure 1 presents an algorithm for performing this exploration. The main data structures used are a *Stack* to hold the states from which the behavior of the system remains to be investigated, and a hash table H to store the states from which the behavior of the system has already been investigated. This algorithm looks like a classical exploration of all possible transition sequences. The difference is that, instead of executing systematically *all* transitions enabled in a state, we choose *only some* of these transitions to be executed. The basic idea for selecting amongst the enabled transitions those that have to be executed is the following. Whenever several independent transitions are enabled at a given state, we execute only one of these transitions. This will ensure that we generate only one interleaving of these independent

transitions. When enabled transitions are dependent, their executions lead to different traces corresponding to different nondeterministic choices which can be made by the system. We then have to execute all these transitions in order to consider all possible traces the system is able to perform. The function **select_transitions** described in Figure 2 performs this selection.

To select amongst the enabled transitions those that have to be executed, the function **select_transitions** uses the notion of *sleep set*. A sleep set is a set of transitions. One sleep set is associated with each state s reached during the search. It is stored onto the stack with s. The sleep set associated with 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 function **select_transitions** uses the sleep set associated to the current state and returns a set of transitions that have to be executed. To each transition t of this set, it "attaches" the sleep set which will be associated to the state $s' = (s \setminus {}^\bullet t) \cup t^\bullet$ reached after the execution of t.

In what follows, two transitions t_1, t_2 are referred to as being in *conflict* iff $({}^\bullet t_1 \cap {}^\bullet t_2) \neq \emptyset$. Checking if two enabled transitions are dependent can be done by checking if they are in conflict. Two transitions t, t' are in *indirect conflict* iff $\exists t_1, \ldots, t_n : ({}^\bullet t \cap {}^\bullet t_1) \neq \emptyset \wedge ({}^\bullet t_1 \cap {}^\bullet t_2) \neq \emptyset \wedge \ldots \wedge ({}^\bullet t_{n-1} \cap {}^\bullet t_n) \neq \emptyset \wedge ({}^\bullet t_n \cap {}^\bullet t') \neq \emptyset$.

The function **select_transitions** performs two distinct tasks: it selects the transitions to be executed and it computes the sleep sets to be passed along with these transitions. We first describe the selection of transitions. Thereafter, we will describe the role of sleep sets and how they are computed. The selection of transitions starts with the enabled transitions that are not in the sleep set associated to the current state. Two cases can occur.

1. There is a transition that is only in conflict or indirect conflict with enabled transitions. Then, this transition and all the transitions that are in conflict or indirect conflict with it are selected. Indeed, these transitions are independent with respect to the rest of the system and their occurrence can not be influenced by it. An interesting special case is that of an enabled transition that is not in conflict with any other transition. In this situation, this transition alone is selected.

2. There is no transition that is only in conflict or indirect conflict with enabled transitions, i.e. each transition is in conflict or indirect conflict with at least one nonenabled transition. In this case all enabled transitions have to be explored. Indeed, let t be an enabled transition that is in conflict with a transition x that is not enabled in the current state (this is a situation of *confusion* [Rei85]). It is possible that x will become enabled later because of the execution of some other transitions independent with t. At that time, the execution of t could be replaced by the execution of x. Thus when we select t, we also have to check if the execution of t could be replaced by the execution of x after the execution of some other enabled transitions independent with t (i.e. to check if the confusion actually leads to a "conflict"): we thus have to select all enabled transitions that are dependent (as usual) and independent with t, i.e. all enabled transitions. Note that, if a confusion does not lead to a "conflict", several sequences corresponding to prefixes of several interleavings of a single trace will be explored.

The selection procedure we have given can lead to *independent* transitions simultaneously being selected. This can cause the wasteful exploration of several interleavings of these transitions. "Sleep sets" are introduced to control this wastefulness. Imagine, for instance, that

```
select_transitions(s,Sleep) {
  T = enabled(s)\Sleep;
  if ∃t ∈ T :conflict(t)⊆ enabled(s) then selection= {t} ∪ (conflict(t)∩T);
  else selection= T;
  result= ∅;
  while selection ≠ ∅ {
     t =one_element_of(selection);
     result=result ∪ attach(t, Sleep);
     Sleep = Sleep ∪ {t};
     selection=selection \{t};
     }
  return(result);
  }
```

Figure 2: Selection Amongst Enabled Transitions

we have a situation of "confusion" with two enabled *independent* transitions a and b, and that a is in conflict with only one nonenabled transition c. We have to explore both the result of transition a and of transition b. Choosing transition a amounts to choosing one interleaving of a and b to be explored. The purpose of exploring the result of transition b is to check if it will eventually enable the transition c with which a is in conflict. So, when doing this, there is no point in exploring the result of transition a. Thus a is introduced in the sleep set that will be associated to the state reached after the execution of transition b. In the remainder of the search starting at that state, a will be only removed from the sleep set when a transition which is in conflict (i.e. *dependent*) with it is selected.

More generally, when independent transitions are selected, the computation of the sleep sets is as follows. One transition is selected first. This transition (let us call it t_1) is the first step in the exploration of *one* interleaving of the transitions. Its sleep set is the current sleep set unmodified except for the elimination of transitions that are dependent with it. When the next transition (t_2) is selected, t_1 is added to its sleep set as long as it is not dependent with it. One then proceeds in a similar way with the remaining transitions. Specifically, when a transition t_i is selected, its sleep set is augmented with all previous transitions (t_j with $j < i$) that are not dependent with it. Note that if only one transition is selected, the sleep set is passed on unmodified.

Let us now see how the procedure we have described is implemented in the algorithm of Figure 2. The function **conflict(t)** returns the transitions that are in conflict and in indirect conflict with t. The function **enabled(s)** returns all enabled transitions in state **s**. The first step is to ignore the transitions that are in the sleep set associated with the current state. If there is a transition t among the remaining enabled transitions such that all transitions in **conflict(t)** are enabled, then t and the transitions of **conflict(t)** that are not in the current sleep set are selected (in practice, if there are several such enabled transitions, one chooses one that minimizes the number of transitions in the **selection** set). The other remaining possibility is that of a case of "confusion" in which all enabled transitions that are not in the sleep set are selected.

The "while" loop then computes the sleep sets that have to be attached to the selected tran-

	Classical Algorithm			New Algorithm		
	Total Run Time (sec)	States	Trans.	Total Run Time (sec)	States	Trans.
rr4	0.7	144	368	0.1	20	24
rr5	1.6	360	1100	0.2	25	30
rr6	4.6	864	3072	0.3	30	36
rr7	12.7	2016	8176	0.3	35	42

Table 1: Analysis of the Round Robin Access Protocol

sitions. For doing this, we use the procedure **attach**(t,*Sleep*) which attaches to the transition t the transitions of *Sleep* that are independent with t, i.e. the set $\{t' \in Sleep : ({}^\bullet t \cap {}^\bullet t') = \emptyset\}$. The first transition to be selected in the "while" loop thus simply inherits the current sleep set minus the transitions that are dependent with it. The sleep set of the following transitions is then constructed from the inherited sleep set augmented with the already selected transitions.

All this ensures that *at least one* interleaving for each trace of the system is explored while still avoiding the construction of all possible interleavings of enabled independent transitions. The practical advantage of the algorithm presented here is that, by construction, the state-graph G' explored by this algorithm is a "sub-graph" of the usual state-graph G representing all possible transition sequences of the system. (By sub-graph, we mean that the states of G' are states of G and the transitions of G' are transitions of G.) Moreover, the function **select_transitions** required for constructing G' can be implemented in such a way that the order of its time complexity is the same as the one of the function **enabled** that is used to construct G. Thus our algorithm never uses more resources than (i.e. has the same worst-case asymptotic complexity as) the classical state space exploration algorithm and is often much more efficient in practice. Of course, if no simultaneous enabled independent transitions are encountered during the search, our method becomes equivalent to the classical one.

Table 1 compares the performance of a classical depth-first search algorithm against the deadlock detection algorithm presented in this section for analyzing the Round Robin Access Protocol described in [GS90] for a ring of 4 (rr4) to 7 (rr7) participants. One clearly sees that our method is much more efficient, *both in time and memory*, than the classical one. With our method, the number of states and transitions grows in a linear way with the number of participants in the protocol.

The algorithm presented in this paper is a simplified and improved version of the one presented in [God90] that was designed to generate at least one interleaving for each possible trace of a concurrent program. The simplification comes essentially from the fact that, when looking for deadlocks, if the algorithm detects that one of the concurrent processes can loop in the current trace being explored, the exploration of this trace can stop without generating the remainder of this trace as it had to be done in [God90]. Hence, the number of explored states and transitions with the new algorithm presented here is always less than or equal to the number explored with the algorithm described in [God90]. Moreover, in the present version, it is no longer necessary to store explicitly the transitions of the state-graph and the additional information that was required in the algorithm of [God90]. Hence the present version always requires less resources.

4 Verifying Safety Properties

In this section, we show how the reachability of a global state, the reachability of a local state, and finally the verification of a safety property can be reduced to deadlock detection. Note that we cannot directly use the algorithm of Section 3 to determine reachability since it usually does not generate all reachable states.

The algorithm presented in the previous section can easily be used to check if a given *global* state $\mathbf{s} = (s_1, s_2, \ldots, s_n)$ of a concurrent program P is reachable. This is done by adding to all A_i's a transition $(s_i, \delta, stop_i)$ for $s_i \in \mathbf{s}$. We call the modified system P_M. Then, the state $\mathbf{s}' = (stop_1, stop_2, \ldots, stop_n)$ is a deadlock of P_M iff the state $\mathbf{s}$ is reachable. Indeed, the synchronization of all processes on the transition δ leading to $\mathbf{s}'$ in P_M is possible only if the global state $\mathbf{s}$ is reachable. Hence checking the reachability of $\mathbf{s}$ amounts to checking if $\mathbf{s}'$ is a deadlock of the modified system P_M.

Let us now turn to the problem of checking the reachability of a given *local* state ℓ. A local state is defined as an incompletely specified global state. In other words, it is a tuple of states of some (but not all) processes. For convenience, we define a local state ℓ as a subset of $\bigcup_i S_i$ such that, for each $1 \leq i \leq n$, $|\ell \cap S_i| \leq 1$. Checking for the reachability of a local state ℓ means checking for the reachability of *some* global state $\mathbf{s} = (s_1, s_2, \ldots, s_n)$ such that $\ell \subset \mathbf{s}$. One can reduce the problem of checking the reachability of a local state ℓ to that of checking the reachability of global states (and hence to deadlock detection) by enumerating all global states $\mathbf{s}$ such that $\ell \subset \mathbf{s}$ and checking if at least one of these is reachable. Unfortunately, this approach is not practical since the number of states $\mathbf{s}$ such that $\ell \subset \mathbf{s}$ can be very large.

We need a more direct reduction. A first step is to apply the construction we used above for transforming global reachability to deadlock detection. Specifically, we add to each automaton A_i that has a state $s_i \in \ell$ a transition $(s_i, \delta, stop_i)$. Let the modified system be P_M. Unfortunately, the reachability of ℓ does not induce a deadlock in P_M since processes that do no have a state in ℓ can still be active. We thus have a form of *livelock*. To simplify the following discussion, we denote by P_ℓ the processes that have a state in ℓ and by $P_{\neg\ell}$ those that do not have such a state.

The next step is to transform the system P_M in such a way that a deadlock rather than a livelock is reached if ℓ is reachable. The idea is to ensure that the processes in $P_{\neg\ell}$ can be blocked whenever the livelock of P_M corresponding to ℓ is reached. For each process $P_i \in P_{\neg\ell}$, we add a transition $(s_i, \delta_i, stop_i)$ to one state s_i of each cycle occurring in the graph A_i of P_i. Also, for each process $P_i \in P_\ell$, we add the transition $(stop_i, \delta_j, stop_i)$ for each j such that $P_j \in P_{\neg\ell}$. Let $P_{M'}$ denote the result of this transformation. It is easy to convince oneself that this transformation does not modify the reachability of ℓ.

The following theorem ensures that the reachability of ℓ can be determined by using our deadlock detection algorithm.

Theorem 4.1 *A local state ℓ of a system P is reachable iff there exists a deadlock $\mathbf{s}'$ in $P_{M'}$ such that $stop_i \in \mathbf{s}'$ for all i corresponding to a process in P_ℓ.*

Sketch of the Proof: First we use the fact that if ℓ is reachable in P it is reachable in $P_{M'}$. Now, if ℓ is reachable in $P_{M'}$, there exists a global state $\mathbf{s}$ such that $\ell \subset \mathbf{s}$ and $\mathbf{s}$ is reachable.

When the system reaches the state s, the processes in P_ℓ can execute the transition δ and reach their state $stop_i$. The other processes may still have the opportunity to evolve. Even if there are processes P_j in $P_{\neg l}$ that are able to remain active, by construction, we know that there exists at least one execution for each of these P_j that will eventually lead to a state where a transition δ_j is possible. This transition is synchronized with the processes in P_ℓ and can occur since the processes in P_ℓ are in their state $stop_i$ where they are ready to synchronize on any transition δ_j. After this synchronization, the process P_j reaches its state $stop_j$ and is blocked for ever. The same will happen for all other processes P_j in $P_{\neg l}$. Hence we reach a global state where all processes are in their state $stop_i$. This is a deadlock state. Indeed, in that state only the processes in P_ℓ have possible transitions in this state and these are synchronized with the processes in $P_{\neg \ell}$ which are blocked.
The other direction of the theorem is immediate to establish. ∎

One may wonder if adding the states $stop_i$ could increase the number of reachable states of the system. An analysis of our reduction shows that this is not the case if we stop the search as soon as a state s such that $\ell \subset s$ is reached.

We can now turn to the verification of safety properties. Safety properties can be represented by prefix closed finite automata on finite words [AS87, BFG+91]. We assume such a representation A_S and proceed as follows:

1. Build the automaton $A_{\neg S}$ corresponding to the complement of A_S. Since A_S is prefix closed, $A_{\neg S}$ is naturally an automaton with only one accepting state (denoted X).

2. Check if the local state X is reachable in the concurrent system composed of the automata A_i, $1 \leq i \leq n$, and of the automaton $A_{\neg S}$.

Note that this framework is still applicable for safety properties represented by more than one communicating automaton.

5 Comparison with Other Work and Conclusions

We have presented a new algorithm for detecting deadlocks. We have shown that this algorithm never uses more resources and can be much more efficient than the classical exploration of the whole state space of the system being checked. This algorithm is a simplified and improved version of the one presented in [God90]. The basic idea of the simplification is close to the one used in [Val88] where Valmari's stubborn set method is adapted to deadlock detection for Petri nets. The advantages of our method are that it can be very easily implemented and that its cost per state explored is comparable to the one incurred when doing an exhaustive exploration of the state space. Our method is also applicable to Petri nets as well as to communicating Petri nets. Finally, our approach to verifying general safety properties by using a deadlock detection algorithm is new.

Our method has the advantages of "on the fly verification", i.e. we compose the program and the property without ever building an automaton representing the global behavior of 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 an explicit representation for the global

behavior of the program to be built. Note that our method is fully compatible with techniques for compactly representing the state space as described in [Hol88, CVWY90].

The main principles of our verification technique can also be profitably used for some applications in the field of Artificial Intelligence. In [GK91], it is shown that the algorithm presented here can be used as a search method suitable for planning the reactions of an agent operating in a highly unpredictable environment.

Acknowledgements

We wish to thank Costas Courcoubetis, Froduald Kabanza, Antti Valmari, Mihalis Yannakakis and anonymous referees for helpful comments on this paper.

References

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

[BFG+91] A. Bouajjani, J.-C. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for branching semantics. In *Proc. 12th Int. Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, Springer-Verlag, July 1991.

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

[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.

[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.

[CVWY90] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Proc. Workshop on Computer Aided Verification*, Rutgers, June 1990.

[FM91] J.C. Fernandez and L. Mounier. On the fly verification of behavioural equivalences and preorders. In *Proc. Workshop on Computer Aided Verification*, Aalborg, July 1991.

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

[GK91] P. Godefroid and F. Kabanza. An efficient reactive planner for synthesizing reactive plans. In *Proceedings of AAAI-91*, volume 2, pages 640–645, Anaheim, July 1991.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proc. Workshop on Computer Aided Verification*, Rutgers, June 1990.

[GS90] S. Graf and B. Steffen. Using interface specifications for compositional reduction. In *Proc. Workshop on Computer Aided Verification*, Rutgers, June 1990.

[GW91] 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.

[Hol88] G. Holzmann. An improved protocol reachability analysis technique. *Software Practice and Experience*, pages 137–161, February 1988.

[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.

[JJ91] C. Jard and T. Jeron. Bounded-memory algorithms for verification on the fly. In *Proc. Workshop on Computer Aided Verification*, Aalborg, July 1991.

[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.

[MP84] Z. Manna and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Science of Computer Programming*, 4:257–289, 1984.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[PL90] D. K. Probst and H. F. Li. Using partial-order semantics to avoid the state explosion problem in asynchronous systems. In *Proc. Workshop on Computer Aided Verification*, Rutgers, June 1990.

[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.

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the AMS*, 141:1–35, 1969.

[Saf88] Shmuel Safra. On the complexity of omega-automata. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, White Plains, oct 1988.

[SVW87] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

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

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

[Val90] A. Valmari. A stubborn attack on state explosion. In *Proc. Workshop on Computer Aided Verification*, Rutgers, June 1990.

[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.