On the Costs and Benefits of using Partial-Order Methods for the Verification of Concurrent Systems

Patrice Godefroid

Proceedings of DIMACS Workshop on Partial-Order Methods in Verification, AMS, Princeton, July 1996.

On the Costs and Benefits of using Partial-Order Methods for the Verification of Concurrent Systems (Invited Paper)

Patrice Godefroid

Abstract. Verification by state-space exploration is one of the most successful strategies for analyzing the correctness of finite-state concurrent reactive systems. Partial-order methods are algorithms for dynamically pruning the state space of such systems without incurring the risk of any incompleteness in the verification results. This paper presents results of experiments performed with these algorithms on real protocol examples, and discusses the practical significance of partial-order methods.

1. Introduction

State-space exploration is one of the most successful strategies for checking the correctness of finite-state concurrent reactive systems. It consists in exploring a global state graph, called the *state space*, representing the combined behavior of all concurrent components in the system. Many different types of properties of a system can be checked by exploring its state space: deadlocks, dead code, unspecified receptions, violations of user-specified assertions, etc. Moreover, the range of properties that state-space exploration techniques can verify has been substantially broadened during the last decade thanks to the development of model-checking methods for various temporal logics (e.g., [**CES86, LP85, QS81, VW86**]).

The main limit of this approach to verification is the often excessive size of the state space. Owing to simple combinatorics, this size can be exponential in the size of the description of the system being analyzed. This exponential growth is known as the *state-explosion problem*. The state-explosion problem is due, among other causes, to the modeling of concurrency by interleaving, or, more accurately, to the exploration of all possible interleavings of concurrent events. For instance, the execution of n concurrent events is investigated by exploring all $n!$ interleavings of these events.

Recently, a collection of verification techniques, referred to as *"partial-order methods"*, have demonstrated that exploring *all* interleavings of concurrent events is not a priori necessary for verification. Indeed, interleavings corresponding to the same concurrent execution contain related information. The intuition behind partial-order methods 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

1991 *Mathematics Subject Classification.* Primary 68Q05, 68Q60; Secondary 03B70.

way of analyzing concurrent executions. Instead, concurrent events should be left unordered since the order of their occurrence is irrelevant. Hence the name "partial-order methods". However, rather than choosing to work with direct representations of partial orders, these algorithms keep to an interleaving representation of partial orders, but attempt to limit the expansion of each partial-order computation to just *one* of its interleavings, instead of all of them. Precisely, given a property φ, partial-order methods explore only a reduced part of the global state space that is provably sufficient to check the given property. The difference between the reduced and the global state spaces is that not all interleavings of concurrent events are systematically represented in the reduced one. In what follows, we call "partial-order method" any algorithm for generating such a reduced state space.

Partial-order methods as defined above first appeared independently in [**Val88a, Val88b**] and [**God90, GW91b**], and were developed further in [**Val90, GW91a, GHP92, HGP92, GP93, Pel93, Val93, WG93, GKPP94, HP94, Pel94**]. A detailed comparison of the results published in these papers is available in [**God96**]. Partial-order methods are now used in several existing verification tools, and have been tested on numerous real-protocol examples (e.g., see [**GHP92, HGP92, HP94, GPS96**]).

Of course, it has been recognized for some time before the early 90's that concurrency and nondeterminism are not the same thing. This observation has actually inspired a fairly large body of work on so-called "partial-order models" of concurrency (e.g., [**Lam78, Maz86, Pra86, Win86**]). Work in this area studies various semantics for concurrency, and compares their properties. Also, partial-order temporal logics (e.g., [**PW84, KP86, KP87, Pen88, Pen90**]) have been designed to be semantically more expressive than previously existing (linear-time and branching-time) temporal logics. In contrast, partial-order methods yields results identical to those of verification methods based on classical interleaving semantics, they just make it possible to perform the verification more efficiently.

Several approximate methods based on simple heuristics have been proposed to restrict the number of interleavings that are explored [**GH85, Wes86, Hol87**]. These heuristics carry with them the risk of incomplete verification results, i.e., they can detect errors but cannot prove the absence of errors. In contrast, partial-order methods reduce the number of interleavings that must be inspected in a completely reliable manner, provably without the risk of any incompleteness in the verification results.

Strategies for proving properties of concurrent systems without considering all possible interleavings of their concurrent actions have been proposed in [**AFdR80, EF82, Pnu85, SdR89, KP92b, JZ93**]. These proof methods are applied in the context of an inference system, in which the correctness of a system is established by proving assertions about its components. This approach to verification has the advantage of not being restricted to finite-state systems. On the other hand, it requires proofs that are manual. Even with the help of a theorem prover, carrying out proofs with a theorem prover is far from fully automatic since most steps of the proof require inventive interventions from the user. In contrast, the focus of the partial-order methods we discuss in this paper is purely on algorithmic issues, since we discuss fully-automatic state-space exploration techniques.

The idea that the cost of modeling concurrency by interleaving can be avoided in finite-state verification also appeared in [**JK90, PL90, McM92, Esp94**]. In [**JK90**], the problem of finding an "optimal" reduced state space with just enough transitions and states to preserve Mazurkiewicz's trace semantics is addressed. In [**PL90**], a method that relies on a pomset grammar description of the system is introduced. Also, in [**McM92, Esp94**], one finds a verification method that works by unfolding a Petri net description of a concurrent system into a finite acyclic structure. These methods are quite different from those discussed in this work. Note that so far none of these other methods have been widely experimented on a large set of realistic examples, as it has been the case for the partial-order methods discussed here.

2. Basic Notions

Consider a concurrent system composed of several processes. Let us assume that the system is represented by a set δ of *system transitions*, specified for instance in some guarded-command modeling language. The choice of a particular modeling language and semantics is not essential for the following discussion. What matters is that it is possible to compute from δ a global transition system A_G (or "global state space") representing the joint behavior of all the processes in the system. For the sake of simplicity, we will assume that each transition of A_G corresponds to the execution of one system transition $t \in \delta$.[1] We will write $s \xrightarrow{t} s'$ to mean that the execution of the transition $t \in \delta$ leads the system from the state s of A_G to the state s' of A_G, and $s \stackrel{w}{\Rightarrow} s'$ to mean that the execution of the sequence $w \in \delta^*$ of transitions leads from s to s'.

The basic idea that enables us to check properties of A_G without constructing the whole of A_G is the following: A_G contains many paths that correspond simply to different execution orders of the same system transitions. If these transitions are "independent", for instance because they are executed by noninteracting processes, then changing their order will not modify their combined effect.

This notion of independency between transitions and its complementary notion, the notion of dependency, can be formalized by the following definition (adapted from [**KP92a**]).

DEFINITION 2.1. Let δ be the set of system transitions and $D \subseteq \delta \times \delta$ be a binary, reflexive, and symmetric relation. The relation D is a *valid dependency relation* for the system iff for all $t_1, t_2 \in \delta$, $(t_1, t_2) \notin D$ (t_1 and t_2 are independent) implies that the two following properties hold for all global states s in the global state space A_G of the system:

1. if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 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 $s \stackrel{t_1 t_2}{\Rightarrow} s'$ and $s \stackrel{t_2 t_1}{\Rightarrow} s'$ (commutativity of enabled independent transitions).

This definition characterizes the properties of possible "valid" dependency relations for the transitions of a given system. Note that it is not practical to check the two properties listed above for all pairs of transitions for all states in order to determine which transitions are independent and which are not. Therefore, in practice, one uses easily checkable syntactic conditions that are sufficient for transitions to be independent. See [**God96**] for a detailed presentation of that topic.

[1] Transitions are assumed to be deterministic: the execution of a transition t in a state s leads to a *unique* successor state. This is not a restriction since "nondeterministic transitions" can always be modeled by a set of deterministic transitions with non mutually exclusive guards.

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. 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. 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 system. The criterion for a partial construction of the state-space would then be 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's 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. With some algorithms, it is even possible to use definitions of independence that are weaker than the one of Definition 2.1 (e.g., [**GP93, God96**]).

3. The Algorithms

In this section, we present the basic algorithmic ideas used in the style of partial-order verification methods we are considering. For simplicity, we only consider the problem of detecting terminating (deadlock) states. In order to check for properties more elaborate than deadlocks (such as arbitrary safety properties or linear-time temporal-logic formulas), it is usually necessary to preserve more information in the reduced state space A_R, i.e, to explore more states and transitions. This is done by enforcing additional conditions that have to be satisfied during the generation of A_R. We refer the reader to [**God96**] for a detailed comparison of the various techniques that have been proposed to address this problem.

The specification of the algorithms we discuss here 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 the partial-order algorithms 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 actually corresponds to a whole family of algorithms [**Ove81, Val91, GW91b, GP93**]. It is shown in [**God96**] that all these algorithms (including Valmari's algorithms for computing "strong stubborn sets") compute *persistent sets*. The second type of technique is the sleep set technique (e.g., [**GW93**]). Interestingly, these two techniques are compatible and can be used simultaneously to further improve the selection of the set T. We first describe persistent-set techniques.

Intuitively, a subset T of the set of transitions enabled in a state s of A_G is called *persistent in s* if all transitions not in T that are enabled in s, or in a state reachable from s through transitions not in T, are independent with 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 [**GP93**].

DEFINITION 3.1. A set T of transitions enabled in a state s is *persistent in s* iff, for all nonempty sequences of transitions

$$s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$$

from s in A_G and including only transitions $t_i \notin T$, $1 \leq i \leq n$, t_n is independent with 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. Persistent sets are very similar, although not equivalent, to the "faithful decompositions" introduced (independently) in [**KP92b**] and to the "ample sets" used in [**Pel93**].

Let a *persistent-set selective search* be a selective search through A_G which, in each state s that it reaches, explores only a set T of enabled transitions that is persistent in s, and that is nonempty if there exist transitions enabled in s. It is easy to prove that a persistent-set selective search started from the initial state of A_G will explore all deadlocks of A_G [**God96**].

Of course, the key element required for the implementation of a persistent-set selective search is an algorithm for computing persistent sets. Such algorithms [**Ove81, Val91, GW91b, GP93**] infer the persistent sets from the static structure (code) of the system being verified. They differ by the type of information about the representation of the system that they use (e.g., "distinguishing between internal 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 [**God96**] for a detailed comparison of these algorithms and of their complexity. 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 in A_R.

The second technique for computing the set of transitions T to consider in a selective search is the sleep set technique [**GW93**] introduced in [**God90**]. 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 [**God96**], which can still be very useful as we will see in Section 6. 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. Again, we refer the reader to [**God96**] for a detailed presentation of the sleep set algorithm and of its complexity.

4. How Can Partial-Order Methods Be Evaluated?

How much can one gain by using these algorithms? It is very difficult to give a general answer. Indeed, one can quite easily construct families of systems for which nothing is gained whatsoever. Examples of such systems are systems where the coupling between the processes is so tight that two independent transitions are never simultaneously enabled. (The system is in fact purely sequential.) In this case, partial-order methods yield no reduction, and the selective search becomes equivalent to a classical exhaustive search.

On the other hand, it is also easy to construct systems for which the growth of the state space when the number of processes in the system increases is reduced from exponential to polynomial by a selective search. This is the case, for instance, for the well-known dining-philosophers example [**Val88a**]. Going one step further, it is also possible to find examples of systems for which the global state space increases in size when the value of some parameter grows, while the reduced state space remains the same. See Chapter 8 of [**God96**] for such an example.

Clearly, by a biased choice of examples, an arbitrarily exaggerated impression of the improvements could thus be suggested. For instance, by setting the number of philosophers to a sufficiently large number, we can claim that we can verify properties of systems with astronomical numbers of states, like 10^{20} states as in [**BCM**$^{+}$**90**], or even systems with infinite numbers of states. Of course, this should be taken with a grain of salt since the fact that checking only a small part of such enormous state spaces is sufficient only indicates that most of the states in the global state space are uninteresting. This observation leads us to the following conclusion: the number of states in the global state space of a system does not give a good measure of its "complexity".

Along the same line of thought, the study of the asymptotic behavior of the function giving the number of states for different numbers of processes in a system is only of limited practical interest. Indeed, state-space exploration techniques are rarely used to verify systems composed of tens of identical processes. For such systems, it is preferable to use other verification techniques specially tailored for proving properties of systems with undefined numbers of participants (e.g., [**KM89, WL89**]).

Consequently, experiments with realistic examples, including industrial-size ones, appear to be the most informative approach to evaluating partial-order verification methods.

5. Evaluation

In order to perform experiments on complex concurrent systems, we have implemented a selective search algorithm using persistent sets and sleep sets in an add-on package for the protocol verification system SPIN [**Hol91**]. SPIN is a verification tool for communication protocols described in the Promela language. Promela is a nondeterministic guarded-command language. Promela defines systems of asynchronously executing concurrent processes that can interact via shared variables and message channels. Interaction via message channels can be either synchronous (i.e., by rendez-vous) or asynchronous (buffered) with arbitrary (user-specified) buffer capacities, and arbitrary numbers of message parameters. These different types of communication can be combined. Given a concurrent system described by a Promela program, SPIN can verify properties of the system by performing a depth-first search in the global state space of the system.

The partial-order package for SPIN that we have developed is available free of charge for educational and research purposes by anonymous ftp from ftp.montefiore.ulg.ac.be in the /pub/po-package directory. More information on the partial-order package can be found in the README file in this directory.

The partial-order package has been tested on various examples of protocols. The aim of these experiments was to determine the type of reduction that can be expected on real protocol examples when using the partial-order verification algorithms, and to evaluate the respective impact of these algorithms on the reduction obtained. In this Section, results obtained with four sample protocols are detailed.

- PFTP is a file transfer protocol presented in Chapter 14 of [**Hol91**], modeled in 206 lines of Promela. It consists of three processes communicating via FIFO channels.
- MULOG3 is a model of a mutual exclusion algorithm presented in [**TN87**], for 3 participants, modeled in 97 lines of Promela. It consists of six processes communicating via FIFO channels and shared variables.
- ABRA is a model of the Abracadabra protocol presented in [**Tur93**], modeled in 168 lines of Promela. It consists of four processes communicating via FIFO channels.
- DTP is a data transfer protocol, modeled in 406 lines of Promela. It consists of three processes communicating via FIFO channels.

We report here experiments performed using four different algorithms.

- DFS denotes an exhaustive search performed in a depth-first order.
- SLEEP denotes a selective search using sleep sets.
- PS denotes a selective search using persistent sets.
- PS+SLEEP denotes a selective search using both persistent sets and sleep sets.

Results of these experiments are presented in Table 1. All experiments were performed on a SPARC2 workstation with 64 Megabytes of RAM, using the Partial-Order Package version 3.0. 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. All visited states are stored in a hash table. To avoid significant run-time penalties for disk-access, visited states can only be stored in randomly accessed memory, i.e., in the main memory available in the computer on which the experiments are performed. Consequently, parameter settings in all the protocols considered were chosen to produce global state spaces that can easily be stored in 64 Megabytes of RAM. For each run, the amount of memory used is directly proportional to the number of stored states.

From the numbers given in Table 1, two main observations can be made concerning the respective impact of persistent sets and sleep sets on the reduction obtained.

- *Persistent Sets yield the most important reductions on the number of visited states. They can also yield good reductions on the number of explored transitions.*
- *Sleep sets yield a less impressive reduction on the number of visited states, but yield very good reductions on the number of explored transitions.*

Protocol	Algorithm	Stored States	Transitions	Time
PFTP	DFS	446,982	1,257,317	478.2
	SLEEP	446,982	622,364	639
	PS	276,722	482,722	662.7
	PS+SLEEP	249,994	351,633	684.7
MULOG3	DFS	38,181	111,668	25.3
	SLEEP	38,181	38,241	30.5
	PS	18,537	38,906	25.8
	PS+SLEEP	17,984	18,057	26
ABRA	DFS	149,816	372,010	494.2
	SLEEP	149,816	176,469	546
	PS	32,289	40,931	166.3
	PS+SLEEP	27,781	34,381	155.7
DTP	DFS	251,409	648,467	200.2
	SLEEP	251,409	269,912	189
	PS	9,904	10,351	11.3
	PS+SLEEP	9,904	10,351	11.5

TABLE 1. Evaluation

For all protocols, the best reductions can be obtained with PS+SLEEP, i.e., by using simultaneously persistent sets and sleep sets. Using persistent sets and sleep sets gives better reductions than using persistent sets alone in almost all cases. For DTP, persistent sets are so good in reducing the number of states and transitions that sleep sets are not able to improve this result.

These results show that using the partial-order methods discussed in this work is basically a no-risk improvement. In the worst case, when the reduction is not sufficient to make up the additional run time overhead (PFTP), the selective search can be slightly slower than a classical search, but the overall time complexity remains linear in the number of explored transitions.

Moreover, using partial-order methods can strongly decrease *both* the time and the memory resources needed to verify properties of concurrent systems (DTP). Therefore, they can be used to verify more complex protocols.

6. State-Space Caching

Another observation that can be made from the results given in Table 1 is the following: when using partial-order methods, and especially when using sleep sets, the number of state matchings, i.e., the number of visited transitions minus the number of visited states, strongly decreases. This phenomenon can be explained as follows [**GHP92**].

When performing a classical search (like DFS), almost all states in the state space of a concurrent system are typically visited several times. There are two causes for this:

1. From the initial state, the explorations of all interleavings of a single finite concurrent execution of the system always lead to the same state. This state will thus be visited several times because of all these interleavings.
2. From the initial state, explorations of different finite concurrent executions may lead to the same state.

When using partial-order methods, and especially when using sleep sets, most of the effects of the first cause given above can be avoided, and, in many cases, most of the states are visited *only once* during the selective search.

States that are visited only once do not need to be stored in memory. Indeed, the only reason why visited states are stored in memory is to avoid redundant explorations of parts of the state space: when a state that has already been visited is visited again later during the search, it is not necessary to revisit all its successors. Unfortunately, it is impossible to determine which states are visited only once before the search is completed. However, if most of the states are visited only once, the probability that a state will be visited again later during the search is very small, and the risk of double work when not storing an already visited state becomes very small as well. This enables one not to store most of the states that have already been visited without incurring too much redundant explorations of parts of the state space. The memory requirements can thus strongly decrease without seriously increasing the time requirements.

State-space caching [**Hol85, JJ91**] is a memory management technique for storing the states encountered during a depth-first search that consists in storing all the states of the current explored path (i.e., those in the current depth-first search "stack") plus as many other states as possible given the remaining amount of available memory. It thus creates a restricted *cache* of selected system states that have already been visited. Initially, all states encountered are stored into the cache. When the cache fills up, old states that are not in the stack are removed from the cache to accommodate new ones. This method never tries to store more states than possible in the cache. Thus, if the size of the cache is greater than the maximal size of the stack during the exploration, the search is not truncated, and eventually terminates.

We have implemented such a caching discipline in our partial-order package. The caching discipline can be used with any of the selective-search algorithms that were considered in the previous section. Results of experiments with different cache sizes and the algorithms DFS, PS, and PS+SLEEP for the MULOG3 protocol are presented in Figure 1. For each run, the run time is directly proportional to the number of explored transitions.

With DFS, these results clearly show that the size of the cache, i.e., the number of stored states, can be reduced to approximately the third of the total number of states in A_G without seriously affecting the number of explored transitions and hence the run time. If the cache is further reduced, the run time increases dramatically, due to redundant explorations of large parts of the state space. This run-time explosion makes state-space caching inefficient under a certain threshold.

With PS, this threshold can be reduced to approximately the eighth of the total number of states. This improvement is not very spectacular because the number of matched states, even when using PS, is still too important (see Table 1). The risk of double work when reaching an already visited state that has been removed from the cache is not reduced enough.

With PS+SLEEP, the situation is different: there is no run-time explosion anymore. Indeed, the number of matched states is reduced so much (see Table 1) that the risk of double work becomes very small. When the cache size is reduced up to the maximal depth of the search (this maximal depth is the lower bound for the cache size since all states of the stack are stored to ensure the termination of the search), the increase of the number of explored transitions is still less than 10%

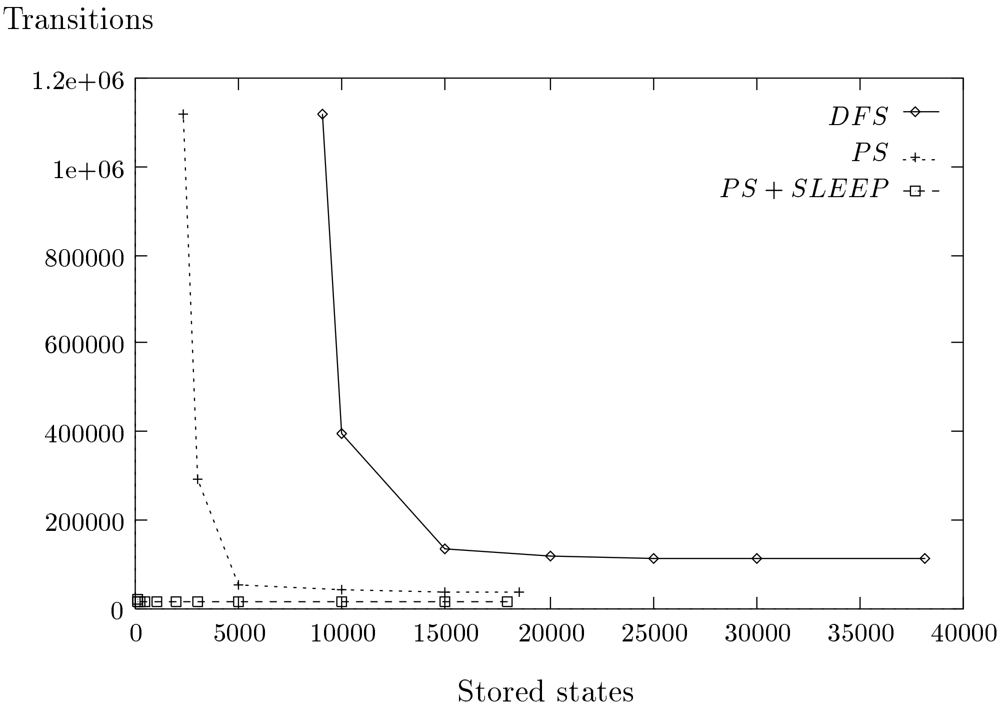

FIGURE 1. Performances of state-space caching for MULOG3

with respect to the number of transitions explored by PS+SLEEP when all visited states are stored in memory, i.e., without using state-space caching.

In other words, the MULOG3 protocol, which has 38,181 reachable states that can be visited by DFS in 25 seconds (see Table 1), can be analyzed with the same run time by using PS+SLEEP and state-space caching while storing no more than 150 states. *The memory requirements are reduced by a factor of 200 while the run time remains the same.*

Of course, the practical interest of this result is that *using partial-order methods and state-space caching together makes possible the complete exploration of very large state spaces*, that could not be explored so far.

For instance, consider two other versions of the MULOG protocol, denoted MULOG4 and MULOG5, with respectively four and five participants. Let PS+SLEEP-+Caching denote a selective search using persistent sets, sleep sets, and state-space caching. Tables 2 and 3 present results of experiments performed on MULOG4 and MULOG5 with the algorithms DFS, PS+SLEEP, and PS+SLEEP+Caching. "Stored states" is the number of stored states at the end of the search. When state-space caching is used, the maximum number of stored states, i.e., the size of the cache, is limited to 300,000 states. (This number is approximately the maximum number of states that can be stored in RAM for MULOG4 and MULOG5 while still avoiding any paging.) "Cleared states" is the number of times a state was removed from the cache. "Matched states" is the number of state matchings that occurred during the search.

Algorithm	Stored St.	Cleared St.	Matched St.	Transitions	Time
DFS	–	–	–	–	–
PS+SLEEP	654,600	0	6,189	660,789	986.4 (2516.7)
PS+SLEEP+Caching	300,000	354,676	6,198	660,874	1122.6 (1184.4)

TABLE 2. Verification of MULOG4

Algorithm	Stored St.	Cleared St.	Matched St.	Transitions	Time
DFS	–	–	–	–	–
PS+SLEEP	–	–	–	–	–
PS+SLEEP+Caching	300,000	28,613,162	349,904	29,263,066	60,633.1

TABLE 3. Verification of MULOG5

For MULOG4, DFS was not able to complete its search, since its global state space is too large to be stored in (64 Megabytes of) memory. Using state-space caching with DFS does not help, because of the run time explosion mentioned above. MULOG4 can still be verified using PS+SLEEP, even without state-space caching. Real time as reported by the UNIX-system time command is given between parentheses below the run time (user time plus system time). The important difference between these two numbers for PS+SLEEP is due to paging (storing 654,600 states of MULOG4 requires more than 64 Megabytes of RAM, so some of them had to be stored on disk).

For MULOG5, the only algorithm that is able to completely verify the correctness of this protocol is PS+SLEEP+Caching. The complete selective search takes approximately 17 hours, and explores 29,263,066 transitions. This means that the *reduced* state space A_R explored by PS+SLEEP contains at most 29,263,066 states. The size of the global state space A_G of MULOG5 is not known, but is very likely several orders of magnitude larger than the largest state spaces that can be explored by other existing verification tools.

Note that the efficiency of the state-space caching technique can be dynamically estimated during the search: if the maximum stack size remains acceptable with respect to the cache size and if the proportion of matched states remains small enough, the run-time explosion will likely be avoided. Else one cannot predict if the cache size is large enough to avoid the run-time explosion.

7. Conclusion

Using partial-order methods is basically a no-risk improvement with respect to a classical exhaustive search in the state space of the system being analyzed. Moreover, partial-order methods can yield substantial improvements in the performances of the verification. Therefore, these methods broaden the applicability of state-space exploration techniques to more complex systems.

The reduction obtained depends on the coupling between the processes in the system. When the coupling is very tight, partial-order methods yield no reduction, and the selective search becomes equivalent to a classical exhaustive search. When the coupling between the processes is very loose, the reduction can be very impressive. For most realistic examples, partial-order methods provide a significant reduction of the memory and time requirements needed to verify protocols.

It is worth noticing that partial-order methods can already yield good performance improvements for verifying systems containing only a handful of processes. It is not necessary to consider systems composed of tens of processes to obtain spectacular reductions. To put it in another way, the part of the state explosion due to the exploration of all possible interleavings of independent transitions can already be very important for systems containing only a few processes, and partial-order methods are able to get rid of most of this explosion.

This very important point emphasizes the practical significance of partial-order methods. Indeed, most of the protocol models that are analyzed with state-space exploration techniques typically contain only a handful of processes. The examples we have considered in Section 5 reflect this reality. For instance, a typical protocol example is usually composed of a few processes that communicate asynchronously by exchanging messages through some communication medium, each process being described by a long piece of sequential code, with complex interactions between control and data.

Not only these systems are very frequent, but they are also very hard to verify: they are complex (several hundreds lines of (Promela) code are needed to model these systems), and their state spaces are highly irregular. Specifically, their state spaces seem to be much more irregular than, for instance, those of systems composed of many identical processes (or pieces of hardware), for which symbolic verification techniques are able to capture the regularity of the state space with the guidance of the user (see, e.g., [**BCM⁺90, McM93**]). In contrast, for examples of the type we are considering here, existing symbolic verification techniques were reported to be inferior to classical state-space exploration algorithms [**HD93**]. Consequently, for this particular, though important, class of systems, partial-order methods are one of the most successful approaches to tackle the state explosion arising during the analysis of such systems.

Finally, we have shown that using partial-order methods, and especially using sleep sets, can substantially improve the state-space caching discipline by getting rid of the main cause of its previous inefficiency, namely prohibitive state matching due to the exploration of all possible interleavings of concurrent executions all leading to the same state. Thanks to sleep sets, the memory requirements needed to verify large *reduced* state spaces can be strongly decreased (several orders of magnitude) without seriously affecting the time requirements. This makes possible the complete exploration of very large reduced state spaces (several tens of million states) in a reasonable time (one night). Used together, partial-order methods and state-space caching significantly push back the limits of verification by state-space exploration.

Note

The results reported in this paper appeared in [**God96**].

References

[AFdR80] K. R. Apt, N. Francez, and W. P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.

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

[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.

[EF82] T. Elrad and N. Francez. Decomposition of distributed programs into communication closed layers. *Science of Computer Programming*, 2:155–173, 1982.

[Esp94] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23:151–195, 1994.

[GH85] M. G. Gouda and J. Y. Han. Protocol validation by fair progress state exploration. *Computer Networks and ISDN systems*, pages 353–361, May 1985.

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

[GKPP94] R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial order approach to branching time model checking. Proceedings of the Third Israel Symposium on Theory of Computing and Systems, 1994.

[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. Extended version in ACM/AMS DIMACS Series, volume 3, pages 321–340, 1991.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, January 1996.

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

[GPS96] P. Godefroid, D. Peled, and M. Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. In *Proceedings of ISSTA'96 (International Symposium on Software Testing and Analysis)*, pages 261–269, San Diego, January 1996.

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

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

[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.

[HD93] A. J. Hu and D. L. Dill. Efficient verification with bdds using implicitly conjoined invariants. In *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 3–14, Elounda, June 1993. Springer-Verlag.

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

[Hol85] G. J. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64(12):2413–2434, 1985.

[Hol87] G. J. Holzmann. Automated protocol validation in argos — assertion proving and scatter searching. *IEEE Trans. on Software Engineering*, 13(6):683–696, 1987.

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

[HP94] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Proc. FORTE'94*, pages 177–191, Bern, 1994.

[JJ91] C. Jard and Th. Jeron. Bounded-memory algorithms for verification on-the-fly. In *Proc. 3rd Workshop on Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, Aalborg, July 1991. Springer-Verlag.

[JK90] R. Janicki and M. Koutny. On some implementation of optimal simulations. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 166–175, Rutgers, June 1990. Springer-Verlag.

[JZ93] W. Janssen and J. Zwiers. Specifying and proving communication closedness in protocols. In *Proc. 13th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, pages 323–339, Liège, May 1993. North-Holland.

[KM89] R. P. Kurshan and K. McMillan. A structural induction theorem for processes. In *Proceedings of the Eigth ACM Symposium on Principles of Distributed Computing*, pages 239–248, Edmonton, Alberta, August 1989.

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

[KP87] S. Katz and D. Peled. Interleaving set temporal logic. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 178–190, Vancouver, August 1987.

[KP92a] S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101:337–359, 1992.

[KP92b] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6:107–120, 1992.

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

[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. Springer-Verlag, 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*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177, Montreal, June 1992. Springer-Verlag.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[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 Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423, Elounda, June 1993. Springer-Verlag.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proc. 6th Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390, Stanford, June 1994. Springer-Verlag.

[Pen88] W. Penczek. A temporal logic for event structures. *Fundamenta Informaticae*, 11(3):297–326, 1988.

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

[PL90] D. K. Probst and H. F. Li. Using partial-order semantics to avoid the state explosion problem in asynchronous systems. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 146–155, Rutgers, June 1990. Springer-Verlag.

[Pnu85] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Proc. Advanced School on Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584, Berlin, 1985. Springer-Verlag.

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

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

[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. Springer-Verlag, 1981.

[SdR89] F. A. Stomp and W. P. de Roever. Designing distributed algorithms by means of formal sequentially phased reasoning. In *Proc. 3rd International Workshop on Distributed Algorithms*, volume 392 of *Lecture Notes in Computer Science*, pages 242–253, Nice, 1989. Springer-Verlag.

[TN87] M. Trehel and M. Naimi. Un algorithme distribué d'exclusion mutuelle en log(n). *Technique et Science Informatiques*, pages 141–150, 1987.

[Tur93] K. J. Turner et al. *Using Formal Description Techniques – An Introduction to Estelle, Lotos and SDL*. Wiley, 1993.

[Val88a] 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.

[Val88b] A. Valmari. Heuristics for lazy state generation speeds up analysis of concurrent systems. In *Proc. of the Finnish Artificial Intelligence Symposium STeP-88*, volume 2, pages 640–650, Helsinki, 1988.

[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 Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 397–408, Elounda, June 1993. Springer-Verlag.

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

[Wes86] C. H. West. Protocol validation by random state exploration. In *Proc. 6th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, pages 233–242. North-Holland, 1986.

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

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

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

Bell Laboratories, Lucent Technologies, 1000 E. Warrenville Road, Naperville, IL 60566, U.S.A.

E-mail address: god@bell-labs.com