State Space Reduction using Partial Order Techniques

E. M. Clarke¹ *, O. Grumberg² **, M. Minea¹, and D. Peled³

¹ Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213-3891, USA
² Dept. of Computer Science, The Technion, Haifa 32000, Israel
³ Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974-2070, USA

October 7, 1997

1 Introduction

One of the main problems in automatic verification of systems is the so-called state space explosion problem. For many types of systems, the number of possible states during system execution grows exponentially with the size of the system and the number of its component parts. This quickly leads to models whose size exceeds the current capabilities of verification tools.

Partial order reduction is a technique that addresses this problem for concurrent asynchronous systems by constructing a smaller state space that is searched by the verification (model checking) algorithms. In general, asynchronous systems are described using an interleaving model of computation. Concurrent events are modeled by allowing their execution in all possible orders relative to each other, creating a large number of possible states and paths. However, specifications typically do not distinguish between all different orders. Partial order reduction considers only a restricted set of behaviors of the system, while guaranteeing that the ignored behaviors do not add any new information.

In this survey we will describe a methods of partial order reduction. The main goal of this paper is to provide an intuitive description of some of the main ideas and presentsome details that can be used for implementing such techniques.

* This research is sponsored by the the Semiconductor Research Corporation (SRC) under Contract No. 97-DJ-294, the National Science Foundation (NSF) under Grant No. CCR-9505472, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-96-C-0071. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of SRC, NSF, DARPA, or the United States Government.

** This research was partially supported by the fund for the promotion of research at the Technion.

Correspondence to: Doron Peled
Reducing the state space by using commutativity between concurrent transitions was suggested by several researchers. In his Ph.D. thesis, Overman [16] suggested a method to avoid exploring all the states of a concurrent system. However, this method was applied to systems without loops. Katz and Peled [12] suggested a proof system for concurrent systems that takes the commutativity between transitions into account. The core of the proof system was based on using proof rules that asserted about sequences that are generated by taking certain subsets of successors from each state.

In the last decade, several researchers have developed techniques for using such reduction principles for model checking. These methods include the stubborn sets method of Valmari [20], the persistent sets method of Godefroid and Wolper [9, 8], and the ample sets method of Peled [18]. These works contain similar ideas, although differ from each other with respect to the details of the suggested reduction. We will follow here the ample sets method.

The name partial order reduction reflects a connection between the initial versions of these reductions and partial order semantics. Roughly, a partially ordered execution is represented by a set of events and a causality relation between them. Some versions of partial order reduction guarantee that the reduced state space will include for each such partially ordered execution there will be at least one linearization (completion into a total order). However, most current methods do not maintain that relation anymore. A different approach that is directly related to partial order semantics is the unfolding method of McMillan [15], which constructs a representation for the partial order executions and use it to detect deadlocks.

2 Principles of Partial Order Reduction

The systems that we analyze are modeled as state transition graphs. If $S$ is the set of states, a transition is a relation $\alpha \subseteq S \times S$, i.e., it can be taken between different pairs of states. A state transition graph is then defined as a tuple $M = (S, S_0, T, L)$, where $S_0 \subseteq S$ is a set of initial states, $T$ is a set of transitions $\alpha \subseteq S \times S$, and $L : S \rightarrow 2^{AP}$ is a labeling function that assigns to each state a subset of some set $AP$ of atomic propositions.

A transition $\alpha \in T$ is enabled in a state $s$ if there exists a state $s'$ such that $(s, s') \in \alpha$ (or in other words $\alpha(s, s')$ holds). If for any state $s$ there is at most one state such that $\alpha(s, s')$, we call $\alpha$ a deterministic transition. In this case we can consider the transition $\alpha$ as a partial function on states instead of a relation and write $s' = \alpha(s)$ instead of $\alpha(s, s')$. The following presentation considers only deterministic transitions, without further explicit mentioning of this aspect.
We will reason about execution sequences of the system, which we call paths. A path in a state-transition graph $M$ is a finite or infinite sequence $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ such that $s_{i+1} = \alpha_i(s_i)$ for every $i$.

As mentioned in the introduction, one of the main reasons for state space explosion in asynchronous systems is that the interleaving model of computation must consider all possible event orderings, in order to avoid the omission of any particular one. However, since the arbitrary ordering imposed by interleaving was introduced to model concurrency in the first place, for independent transitions this ordering is often irrelevant. On the other hand, depending on the specification language, it is possible that the property to be verified is actually able to discriminate between behaviors that only differ by this ordering. To be able to use partial order reduction, it is necessary to have a specification that does not distinguish between such behaviors and a procedure that selects a set of behaviors that constitutes the reduced model. If some behavior is not included in the reduced model, an equivalent one has to be included in order to guarantee correctness.

To illustrate the importance of reduction, consider a system composed of $n$ concurrent processes, $P_1$ through $P_n$. Each process $P_i$ has a transition $\alpha_i$ enabled in some local state $s_i$, that changes the value of the labeling function: $\alpha_i(s_i) = s'_i, L(s_i) = \emptyset, L(s'_i) = \{p\}$, for some $p \in AP$. The concurrent transitions $\alpha_i$ can be ordered in $n!$ possible ways, producing a total of $2^n$ different states, yet it is possible that the specification only needs to establish a property that links the initial global state $(s_1, \ldots, s_n)$ with the resulting state $(s'_1, \ldots, s'_n)$, irrespective of the path taken between these. In this case, it is much more efficient to consider only one particular ordering and the corresponding $n + 1$ states.

Typically, the reduced model is constructed by performing a modified depth-first search on an explicit state representation of the system. Model checking is done in a separate phase, on the resulting reduced state transition graph. It is also possible to construct the reduced model on the fly, while performing model checking. Other variations are to use breadth-first search instead of depth-first search, or to combine partial order reduction with symbolic model checking. A common point for all variants is that the reduced state space is constructed directly, without ever building the full state graph. This would be counter to the purpose of reduction, since it is likely that the full state graph is too large to be constructed in the first place.

Consider, for the purpose of illustration, the case of depth-first search. A typical search that constructs the entire reachable state space would follow all transitions enabled at the current state in the search. With partial order reduction, only a subset of the enabled transitions is expanded at each state $s$. We will call this set $\text{ample}(s)$.

To apply this method, we need a procedure to compute a suitable set $\text{ample}(s)$ for every state $s$. On one hand, $\text{ample}(s)$ has to be significantly smaller than $\text{enabled}(s)$,
in order for the reduction to yield a sizeably smaller state graph. On the other hand, the reduction must be first of all correct, and therefore ample(s) has to include enough transitions such that for each behavior in the full state graph there is one equivalent behavior in the reduced state graph. Finally, computing an ample set should be done with a reasonably small overhead so that verification time is not increased compared to full state space search.

Since the key issue in partial order reduction is to select only a restricted number of orderings between transitions for analysis, the concept of transitions that can be reordered has to be formalized. This can be done by defining the key concept of independence relation between transitions. Two transitions \( \alpha, \beta \in T \) are independent if they satisfy the following two conditions for each state \( s \in S \):

- **Enabledness** If \( \alpha, \beta \in \text{enabled}(s) \) then \( \alpha \in \text{enabled}(\beta(s)) \) and \( \beta \in \text{enabled}(\alpha(s)) \).
- **Commutativity** If \( \alpha, \beta \in \text{enabled}(s) \) then \( \alpha(\beta(s)) = \beta(\alpha(s)) \).

The enabledness condition expresses the fact that two independent transitions that are enabled at a given state cannot disable each other. Note that the weaker definition given here allows independent transitions to enable one another. The commutativity condition states that the execution of two independent transitions in any order (which is guaranteed to be possible by the enabledness condition) leads to the same state. Two transitions are called dependent if they are not independent.

![Independent transitions](image)

**Fig. 1.** Independent transitions

Consider the simple portion of a state transition graph depicted in Figure 2. If transitions \( \alpha \) and \( \beta \) are independent, a possible reduction would be consider only the execution sequence \( s \xrightarrow{\alpha} s_1 \xrightarrow{\beta} s' \) and not the path \( s \xrightarrow{\beta} s_2 \xrightarrow{\alpha} s' \). However, this reduction may not be necessarily correct, either because the checked property can distinguish between the intermediate states \( s_1 \) and \( s_2 \), or because eliminating one of these states may cause some of its successors (which are significant for verification) not to be explored. Additional conditions for the correctness of the reduction are needed, and they will be described in the following.

To address the first of these two issues, we define what it means for a specification to distinguish between two states, by introducing a second key concept, of invisible
transitions. Recall that \( L : S \rightarrow 2^{AP} \) is the labeling function that assigns to each state a set of atomic propositions. The specification does not necessarily refer to the entire set of atomic propositions; let \( AP' \subseteq AP \) be the subset of atomic propositions referenced in the specification. We call a transition \( \alpha \) invisible with respect to some subset \( AP' \subseteq AP \) if its execution between any two states does not change the labeling with atomic propositions from \( AP' \). Formally, the transition \( \alpha \in T \) is invisible with respect to \( AP' \) if for any two states \( s, s' \in S \) such that \( s' = \alpha(s) \) we have \( \overline{L}(s) \cap AP' = \overline{L}(s') \cap AP' \). A transition is visible if it is not invisible. If the subset of atomic propositions \( AP' \) is clear from the context (it is usually the set of atomic propositions contained in the specification), we will simply say that a transition is visible or invisible without explicitly mentioning that this is with respect to \( AP' \).

In asynchronous systems, the number of transitions occurring between two events has no direct relationship to the time delay between them. Furthermore, transitions which are concurrent in the system appear serialized in some order in the interleaving model. These observations argue for a specification which cannot distinguish between sequences of identically labelled states on an execution path of the system.

We call two infinite paths stuttering equivalent if they have identical state labelings after collapsing any finite sequence of identically labeled states to a single state in each of them. In other words, two infinite paths \( \sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \) and \( \rho = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \ldots \) are stuttering equivalent if one can define two infinite sequences of integers \( 0 = i_0 < i_1 < \ldots \) and \( 0 = j_0 < j_1 < \ldots \) such that \( \forall k \geq 0, L(s_{i_k}) = L(s_{i_k+1}) = \ldots = L(s_{i_k+1}) = L(r_{j_k}) = L(r_{j_k+1}) = \ldots = L(r_{j_k+1}) \). The indices \( i_k \) and \( j_k \) are the starting points of identically labeled subsequences of states in the two paths, respectively. We denote the stuttering equivalence relation between \( \sigma \) and \( \rho \) by \( \sigma \sim_{st} \rho \).

![Fig. 2. Stuttering equivalent paths](image)

The temporal logic LTL [6] allows assertions about the temporal behavior of a program. Given a finite set of propositions \( AP \), the LTL formulas are defined inductively as follows:

- every member of \( AP \) is a formula,
- if \( \varphi \) and \( \psi \) are formulas, then so are \( \neg \varphi \), \( \varphi \land \psi \), \( \bigcirc \varphi \) and \( \varphi U \psi \).
An interpretation of an LTL formula is an infinite word \( \xi = x_0 x_1 \cdots \) over the alphabet \( 2^{AP} \), i.e., a mapping from the naturals to \( 2^{AP} \). We write \( \xi_i \) for the suffix of \( \xi \) starting at \( x_i \). The semantics of LTL is as follows:

- \( \xi \models A \) iff \( A \in x_0 \), for \( A \in AP \),
- \( \xi \models \neg \varphi \) iff not \( \xi \models \varphi \),
- \( \xi \models \varphi \land \psi \) iff \( \xi \models \varphi \) and \( \xi \models \psi \),
- \( \xi \models \square \varphi \) iff \( \xi_1 \models \varphi \),
- \( \xi \models \varphi U \psi \) iff there is an \( i \geq 0 \) such that \( \xi_i \models \psi \) and \( \xi_j \models \varphi \) for all \( 0 \leq j < i \).

Let \( false \) as an abbreviation for \( A \land \neg A \), and \( true \) be an abbreviation for \( \neg false \). We also use the following abbreviations: \( \varphi \lor \psi = \neg((\neg \varphi) \land (\neg \psi)) \), \( \Box \varphi = true U \varphi \), \( \bigcirc \varphi = \neg \Box \neg \varphi \).

Given a state transition graph \( M \) and an LTL formula \( \varphi \), the model checking problem for \( M \) and \( \varphi \) is to verify that for every initial state \( s_0 \in S_0 \) and every path \( \xi \) starting in \( s_0 \), it is true that \( \xi \models \varphi \). If this holds, we write \( M \models \varphi \).

An LTL formula \( \varphi \) is invariant under stuttering if for any two paths \( \sigma \) and \( \sigma' \) such that \( \sigma \sim_{st} \sigma' \), we have \( \sigma \models \varphi \) iff \( \sigma' \models \varphi \).

In general, an LTL formula can be sensitive to stuttering if it contains the next-time operator \( \bigcirc \). Denote by LTL\(_{\neg X} \) the subset of logic LTL that does not use the next-time operator. Peled and Wilke show [19] that an LTL property is invariant under stuttering iff it can be expressed in LTL\(_{\neg X} \).

The notion of stuttering equivalence can be extended from paths to state transition graphs. Two state transition graphs \( M \) and \( M' \) are stuttering equivalent iff the following two symmetric conditions hold:

- for each path \( \sigma \) from an initial state of \( M \) there is a path \( \sigma' \) from an initial state of \( M' \) such that \( \sigma \sim_{st} \sigma' \).
- for each path \( \sigma' \) from an initial state of \( M' \) there is a path \( \sigma \) from an initial state of \( M \) such that \( \sigma' \sim_{st} \sigma \).

From the definition of stuttering equivalence of state transition graphs and the theorem about stuttering invariance of LTL\(_{\neg X} \) formulas, one can deduce the following result

If \( M \) and \( M' \) are state transition graphs which are stuttering equivalent, then for any LTL\(_{\neg X} \) property \( \varphi \), \( M \models \varphi \) iff \( M' \models \varphi \).

This result justifies the use of partial order reduction by virtue of the fact that it produces a structure that is stuttering equivalent to the original state transition graph:

\section{Partial Order Reduction for LTL\(_{\neg X} \)}

We have seen in the previous section that the properties of independence and invisibility for transitions and stuttering invariance for LTL\(_{\neg X} \) formulas allow us to verify the
specification for the given system on a reduced model, and thus avoid the generation of all states. The reduced model is constructed by selecting at each step a subset \( \text{ample}(s) \) of the transitions which are enabled at the current state \( s \). We still need a procedure that will determine a suitable set of ample transitions at each state. Rather than directly give an algorithm that solves this problem, in this section we will characterize the set \( \text{ample}(s) \) using a set of conditions. The next section continues by describing various heuristics that can be used to find ample sets that satisfy these conditions.

The first condition is trivial and guarantees that the search algorithm with reduction will make progress if the normal search algorithm would:

**C0 Emptyness** \( \text{ample}(s) = \emptyset \) iff \( \text{enabled}(s) = \emptyset \).

The next constraint is introduced to ensure that any path that is not included in the reduced state-transition graph can be transformed, based on the properties of independent transitions, into a path in the reduced model, and therefore the reduction does not omit any paths which are essential for verification.

**C1 Ample decomposition** In the full state graph, on any path starting from some state \( s \), a transition dependent on a transition from \( \text{ample}(s) \) cannot appear before some transition from \( \text{ample}(s) \) is executed.

To analyze the implications of **C1**, consider an arbitrary sequence of transitions \( \sigma = \alpha_0, \alpha_1 \ldots \) that can be taken from some state \( s_0 \) in the full state transition graph. We outline the basic ideas of a construction that can be used to generate a path in the reduced model that contains all transitions from \( \sigma \). More details of the construction and a proof for its correctness are given by Clarke, Grumberg and Peled [3].

(a) if \( \alpha_0 \in \text{ample}(s_0) \), then \( \alpha_0 \) can be taken from \( s_0 \) in the reduced model, and the path prefix \( s_0 \xrightarrow{\alpha_0} s_1 \) belongs to the reduced model. The construction is continued inductively from \( s_1 \).

(b) if \( \alpha_0 \not\in \text{ample}(s_0) \), consider first the case where the transition sequence \( \sigma \) contains some transition from \( \text{ample}(s_0) \). Let \( \beta \) be the first such transition appearing in \( \sigma \), i.e. \( \beta = \alpha_k \), with \( k \geq 1 \). Then by condition **C1** all transitions \( \alpha_i \) with \( 0 \leq i < k \) must be independent of \( \beta \). Independence implies commutativity, therefore the transition sequence \( \beta\alpha_0\alpha_1\ldots\alpha_{k-1}\alpha_{k+1}\ldots \) is also a transition sequence enabled in \( s_0 \) in the original model. Moreover, since \( \beta \in \text{ample}(s_0) \), the first transition can also be taken in the reduced model and the construction continues form \( s_1 = \beta(s_0) \).

(c) if \( \alpha_0 \not\in \text{ample}(s_0) \) and the sequence \( \sigma \) does not contain any transition from \( \text{ample}(s_0) \), let \( \beta \) be an arbitrary transition from \( \text{ample}(s_0) \). By condition **C1**, none of the transitions in \( \sigma \) can be dependent on \( \beta \). Therefore, if \( s_1 = \beta(s_0) \), the path \( s_0 \xrightarrow{\beta} s_1 \) belongs to the reduced model and the transition sequence \( \sigma \) is still enabled in \( s_1 \).

Transforming a path \( \sigma \) in the full state transition graph into a path \( \sigma' \) including all transitions from \( \sigma \) and with a prefix that belongs to the reduced model is not sufficient.
We still need to know that the constructed path is stuttering equivalent to the initial one, so that the truth value of the specification will not be affected.

**C2 Invisibility** *If a state s is not fully expanded, every transitions α ∈ ample(s) has to be invisible.*

To see the effect of this condition, consider cases (b) and (c) discussed previously in conjunction with C1 (there is no need to discuss (a), since it did not imply any change to the path). For case (b), let \( s_i \) be the states on the initial path \( σ (s_i \xrightarrow{α} s_{i+1}) \) and let \( s'_i = β(s_i) \). Since \( β \) commutes with \( α_i \) for \( i < k \), it follows that \( s'_0 s'_1 \ldots s'_{k-1} s_{k+1} \) is exactly the state sequence obtained by executing the transition sequence \( β α_0 \ldots α_{k-1} \). Since \( β \in ample(s_0) \) but \( α_0 \notin ample(s_0) \), it follows by C2 that \( s_0 \) is not fully expanded, therefore \( β \) has to be invisible. Consequently, \( L(s'_i) = L(s_i) \) \( \forall i \leq k \) and therefore the sequences \( s_0 s_1 \ldots s_k s_{k+1} \ldots \) and \( s_0 s'_0 s'_1 \ldots s'_{k-1} s_{k+1} \ldots \) are stuttering equivalent. Case (c) is similar: Here too, \( β \) has to be invisible and \( L(β(s_i)) = L(s_i) \) for any \( i \), therefore the two state sequences are stuttering equivalent.

Together, C1 and C2 still do not guarantee that a stuttering equivalent path in the reduced model can be found for any path in the original model. To see this, we note that the recursive condition we have outlined is not guaranteed to produce a path that contains all transitions in the original path \( σ \). For case (c), none of the original transitions in \( σ \) is “consumed”. Instead, an auxiliary transition \( β \in ample(s_0) \) is appended to the beginning of \( σ \). If \( σ \) does not contain any ample transition from \( β(s_0) \) either and this step is repeated sufficiently often, a cycle consisting of inserted ample transitions will be closed. Therefore, the expansion of state \( s_0 \) will terminate without ever considering the transition \( α_0 \) which is enabled in that state. If \( α_0 \) is a visible transition, the specification may have different truth values in the original and reduced models. To avoid this case, it is necessary to introduce a third condition:

**C3 Cycle closing condition** *There is no cycle that contains a state s and a transition α enabled in s, such that α does not belong to the ample set of any state on the cycle.*

With C3, the set of conditions that have to be satisfied by ample sets is complete. The next section shows how sets of transitions satisfying these conditions can be computed in practice.
4 Calculating Ample Sets

To obtain an efficient reduction procedure, it is necessary to determine values for the ample sets at each state that result in a significantly smaller number of successor states and are at the same time easy to compute with small overhead. We describe in the following how this can be done for each of the given conditions.

Checking that an ample set is nonempty (condition C0) is trivial. Likewise, to verify condition C2 it suffices to examine each transition in the ample set of a state. Condition C1 however, is more difficult. One reason is that C1 is stated as a property of the full state transition graph, which the reduction technique attempts to avoid in the first place. Second, the formulation of C1 refers to future states that may not have been yet examined in the search. It can be shown [3] that in general checking C1 is at least as hard as checking reachability for the full state transition graph.

Consequently, rather than using an algorithm that needs to check C1 for an arbitrary set of transitions, in general we will exploit the structure of the system to produce sets of ample transitions for which condition C1 is guaranteed to hold. The following exposition discusses such algorithms for two different classes of concurrent systems. Common to both is the modeling of the system as a set of processes, each with a set of transitions (that may be common in several processes). A process has a set of local variables that can be changed only by transitions performed by that process. These variables are part of the local state of the process, and the product of the local states forms the global state of the system. A transition that only changes the local variables of a process is called an internal transition.

The synchronous communication model requires the sender and the receiver to coordinate, such as for instance in Communicating Sequential Processes [10] or the rendezvous model of ADA. Since the sending and the receiving transitions happen simultaneously, they can be considered as a common transition shared by the two processes. We call such a transition a communication transition. Assume for the following that a system has only local and synchronous communication transitions. When a process $P_i$ arrives at a communication point (a send or receive action), the corresponding communication transition is enabled by $P_i$. It will only be enabled globally when the communication partner of $P_i$ arrives at its corresponding transition point. A communication transition between two processes $P_i$ and $P_j$ is said to be locally enabled by $P_i$ at state $s$ if it can be executed from some state $s'$ that has the same local state of $P_i$ as $s$.

A conservative definition of the dependence relation considers all local transitions within a process to be pairwise dependent. Therefore, a communication transition will be dependent on local transitions from both processes. A suitable selection of ample sets that will satisfy C1 is the following:
For a state $s$, select a subset $P$ of processes, such that for any $P_i \in P$ there is no communication locally enabled by $P_i$ with a process outside of $P$. Then select all transitions enabled in state $s$ and belonging to some process in $P$ as the set $ample(s)$.

The partitioning of the processes in two sets guarantees that by executing transitions outside in the ample set it is not possible that a transition dependent on an ample transition will become globally enabled therefore executed before any transition in the ample set. This is exactly the constraint imposed by C1. The new rule can be applied in practice by initially selecting $P$ to consist of a single process. If the constraint is not satisfied, other processes are added to $P$ until it holds. In the worst case, $P$ is the set of all processes, which corresponds to a fully expanded state and no reduction at all.

In the asynchronous communication model, in addition to the local variables of each process, there are shared message queues through which the communication between processes is performed. A process executing a send operation does not have to wait for the destination process to execute the appropriate receive, unless the message queue that would be used for communication is full (we assume a finite-state system and therefore finite queues). Likewise, a receiving process consumes a message from its input queue and does not block unless this queue is empty. We call a send and a receive operation matching if they share the same message queue. For this discussion we assume the existence of a message queue for each pair of communicating processes; different rules can be given for other cases (e.g. one input queue per process).

Note that a send operation can enable a matching receive (if the queue was previously empty) and conversely, a receive operation can enable a matching send (if the queue was previously full). Both of these scenarios are allowed for the definition of transition independence, which merely requires that transitions do not disable each other. Consequently, we consider the following constraint:

For a given state $s$, select as the set $ample(s)$ the set of all enabled transitions from a set of processes $P$ satisfying the following two conditions:

- No process in $P$ has a send transition that is blocked only because the corresponding queue is full.
- No process in $P$ has a receive transition that is blocked only because the corresponding queue is empty.

The above first condition guarantees that the following scenario does not happen:

A sequence of transitions from some processes that are not in $P$ is executed from the current state. Among these transitions, which are independent of those selected, there is eventually a receive transition $\gamma$. Its execution enables a send transition $\alpha$ of some process in $P$, as its queue is no longer full. But $\alpha$ is dependent on some transition in the selected ample set, contradicting C1.
With the first condition above, such a transition $\gamma$ cannot exist. A similar scenario that justifies the second condition can also be shown.

As in the previous case, we can implement this rule by starting with $\mathcal{P}$ consisting of an arbitrary process $P_i$ and selectively adding other processes until the communication conditions are satisfied.

The original formulation of C1 requires knowledge about transitions that can be executed in the future. Consequently, additional information about which transitions may be enabled in the future, which allows more flexibility in choosing ample sets. For instance, the condition given above for the case of synchronous communication can be weakened. A process from $\mathcal{P}$ is allowed to have a locally enabled communication transition with a process outside $\mathcal{P}$ if one can determine that this communication cannot actually take place in any state reachable from the current state.

However, checking that from a given state a transition is disabled in the future is as hard as the model checking problem itself. Again, the solution is to use an analysis that will identify some of the transitions that can no longer become enabled starting from the current state, rather than all of them. This can be done performing a separate reachability analysis for each process, taking advantage of the fact that the state space of a single process is much smaller than the global state space. In the example given above for synchronous communication one could check whether the matching communication transition can be reached in the other process starting from its local state. This analysis assumes that all communication transitions which are joint with other processes are enabled by those processes, and is therefore conservative. Moreover, it is also possible to ignore data values (selectively or completely) and perform in the simplest case only a static analysis of the control flow graph of the process.

This search can be done in a preliminary stage of the reduction algorithm. It can identify unreachable transitions among those transitions with possible dependencies (synchronous/asynchronous communication, use of global variables, etc.). During the subsequent state search, this information can be used to identify more subsets as ample sets and thus improve the efficiency of the reduction.

5 Experience with Partial Order Reduction

Various systems that include of partial order reduction algorithms have been implemented. Our experience is mainly related to the SPIN implementation described by Holzmann and Peled [11]. In this section we will describe various lessons learned about the difficulty of implementation and the efficiency of the partial order reduction.

One noticeable fact about partial order reduction is that it is usually given as a set of principles rather than an algorithm, which calls for an open-ended list of heuristics.
In particular, condition C1 can be satisfied by a trivial implementation that selects all the enabled transitions from any given state. A better implementation performs an analysis which is based e.g., on the types of transitions that are enabled or disabled from the current state. By making the analysis more involved, and taking more cases into consideration, one can obtain a better reduction. On the other hand, the overhead may also grow when using more complicated analysis.

In the SPIN implementation [11], the initial decision was to provide an adequate reduction for the asynchronous communication case. When systems that use other concurrency mechanisms such as shared variables or synchronous communication are verified, the reduction might be far from optimal. Even with this restriction, the size of the SPIN code roughly doubled when the partial order reduction was first added to it.

The effectiveness of the reduction obtained using the partial order techniques varies considerably among different examples. It is immediately observable that the method works only for asynchronous systems, where commutativity between concurrent transitions can be exploited. Thus, for most hardware systems, which are usually synchronous, there is little, if any reduction. Concurrent systems, and in particular distributed programs, which exhibit a lot of parallelism and independency, are the main focus of the partial order reduction. For certain examples, e.g., the distributed sieve of Eratosthenes for calculating prime numbers, a version of concurrent sorting and a leader election in a ring of processes, the reduction was shown to grow exponentially with the number of processes. Some other, more typical examples show reduction by one order of magnitude.

A factor that greatly influences the effectiveness of the reduction is based on the following observation: when two transitions that are independent can both change the truth value of predicates that appear in the checked property (i.e., the transitions are visible), the order between them becomes relevant, even if they may be independent. Specifically, Condition C2 forces visible transitions to be selected into an ample set only with all other visible transitions. A closer look shows that as a result, if an execution is not present because of the reduction, another execution with the same order of visible transitions will be present in the reduced state graph. Experimentally, the effectiveness of the reduction diminishes quickly with the number of predicates used in the specification. For this reason, it is useful to try to simplify the checked properties, e.g., checking separately for $\diamond p$ and for $\diamond q$, rather than $\diamond p \land \diamond q$.

Combining different model checking techniques does not always follow trivially from the separate methods. Other reduction techniques may be more effective in cases where partial order methods fail. Composing methods may result in a way of tackling more diverse systems, or even obtaining a bigger reduction than can be achieved by each one
separately. Partial order reduction has been combined with various other model checking methods as follows:

- Partial order can be performed with 'on-the-fly' model checking [14], i.e., generating the reduced state space at the same time as checking for counterexamples for the checked property. The construction can result in such an example before completing the generation of the entire state graph [18, 20].

- Symmetry reduction can be used to obtain a smaller state space when dealing with systems of identical components. Partial order reduction and symmetry were combined in [5].

- Symbolic model checking is a very effective method, which uses the BDD data structure for manipulating and storing the states. Although it is generally used for synchronous hardware systems, where partial order reduction is not effective, symbolic model checking was shown to give very good results for asynchronous systems, including software as well. One way of combining this method with the partial order reduction was suggested in [1], where a reduction based on breadth first search was used [2]. A different way of combining these methods is suggested in [13], based on statically resolving possible cycles of the constructed reduced state space.

Partial order reduction can also be applied to branching temporal logic and process algebra [7], and be used under fairness assumptions [17]. Ongoing research that is connected to partial order reduction seeks improved version of the reduction, reduction for additional formalisms, and specification formalisms for which reduction is more effective.

References


time logic model checking. In Proceedings of the 3d Israel Symposium on the Theory of
Submitted for journal publication.
8. P. Godefroid and D. Pirotin. Refining dependencies improves partial-order verification
11. G. J. Holzmann and D. Peled. An improvement in formal verification. In Formal Description
In Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for
Concurrency, number 354 in LNCS, pages 489–507, Noordwijkhout, The Netherlands,
software context. In Proc. 1987 IEEE Int. Conf. on Comput.-Aided Design, San Jose, CA,
15. K. McMillan. Using unfolding to avoid the state explosion problem in the verification of
asynchronous circuits. In G. Bochmann and D. Probst, editors, Proc. 4th Workshop on
Comput.-Aided Verification, volume 663 of LNCS, pages 164–177, Montreal, Canada, June
University of California at Los Angeles, 1981.
17. D. Peled. All from one, one for all: on model checking using representatives. In Courcoubetis
[4], pages 409–423.
Dill, editor, Proc. 6th Workshop on Comput.-Aided Verification, volume 818 of LNCS,
19. D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the
20. A. Valmari. A stubborn attack on state explosion. In E. Clarke and R. Kurshan, editors,
Proc. 2nd Workshop on Comput.-Aided Verification, volume 531 of LNCS, pages 156–165,