Static Partial Order Reduction

R. Kurshan[1], V. Levin[1], M. Minea[2], D. Peled[1,2], H. Yenigün[1]

[1] Lucent Technologies, Bell Laboratories, Murray Hill, NJ 07974
[2] Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

Abstract. The state space explosion problem is central to automatic verification algorithms. One of the successful techniques to abate this problem is called 'partial order reduction'. It is based on the observation that in many cases the specification of concurrent programs does not depend on the order in which concurrently executed events are interleaved. In this paper we present a new version of partial order reduction that allows all of the reduction to be set up at the time of compiling the system description. Normally, partial order reduction requires developing specialized verification algorithms, which in the course of a state space search, select a subset of the possible transitions from each reached global state. In our approach, the set of atomic transitions obtained from the system description after our special compilation, already generates a smaller number of choices from each state. Thus, rather than conducting a modified search of the state space generated by the original state transition relation, our approach involves an ordinary search of the reachable state space generated by a modified state transition relation. Among the advantages of this technique over other versions of the reduction is that it can be directly implemented using existing verification tools, as it requires no change of the verification engine: the entire reduction mechanism is set up at compile time. One major application is the use of this reduction technique together with symbolic model checking and localization reduction, obtaining a combined reduction. We discuss an implementation and experimental results for SDL programs translated into COSPAN notation by applying our reduction techniques. This is part of a hardware-software co-verification project.

1 Introduction

One common method for dealing with the intrinsically intractable computational complexity of model-checking asynchronous systems is *partial order reduction.* This reduction technique exploits the common practice of modeling concurrent events in asynchronous systems as an interleaving of the events in all possible execution orders. An important observation about such systems is that the properties one needs to check often do not distinguish among these different orders. The reduction algorithm produces a state graph which contains only a subset of the states and transitions of the original system, but which contains enough information about the modelled system so that it is possible to apply model checking algorithms to it instead of the full state graph. The verified property is guaranteed to be true in the reduced model if and only if it is true in the original model.

Since partial order reduction is naturally defined for asynchronous systems, it has thus far been applied mainly to the verification of software. Traditional partial order reduction algorithms use an explicit state representation and depth first search. In contrast, other techniques for model checking, most notably symbolic model checking based on binary decision diagrams (BDDs), have proved most effective for synchronous systems, in particular for verifying hardware. In this paper we describe a reduction algorithm that was developed to satisfy the following goals:

- Perform efficiently for a system that combines software and hardware, and therefore combine well with symbolic model checking.
- Be independent of the type of search, e.g., be applicable to depth- or breadth-first search without a change.
- Allow a large (in our case, in fact the entire) part of the reduction to be done during compilation of the modelled system.
- Be compatible with existing model checking tools without requiring a change to their search engines.

We show a partial order reduction algorithm that achieves these goals. Our method was motivated by an interest to verify embedded systems containing both hardware and software. In our case, the software was written in the SDL language [13] and was translated into the specification language S/R, which is an automata based language for specifying coordinating processes and used as the input language for the model-checking tool COSPAN [7]. COSPAN runs on synchronous input models and supports a symbolic (BDD-based) state space search.

Previous implementations of partial order reduction algorithms in pre-existing state space search engines required considerable changes in the search mechanism [6, 9]. The alternative was to construct a special tool for performing the search [14]. In [9], a reduction that is based on doing a large part of the calculations at compile time is described. However, some changes are still made to the search engine, controlling the backtracking mechanism in the depth-first search performed in the SPIN [8] model checker.

Since we began with an efficient translator [2] from SDL to S/R, we aimed to investigate whether one could obtain an efficient partial order reduction algorithm with *all* of the reduction-specific calculations taking place at compile time, without changing the search mechanism of the COSPAN verification engine. COSPAN should treat the translated model as a regular model, without having to undergo any changes to implement the reduction. The process is illustrated in Fig. 1. The usual model checking procedure involves translating the source code into some intermediate code, which is then analyzed by the actual model checking algorithms. The reduction usually requires a change to the code of the model checker. In this paper, we suggest a modified partial order reduction that can be applied to the translation. The model checking tool remains unchanged.

A related recent result is the combination of partial order reduction with symbolic model checking reported in [1]. This is based on performing a reduction which uses breadth first search [3]. The method suggested in the present paper is more general in the sense that it is independent of the type of search used (e.g., breadth first or depth first). It can also be applied to existing model checking tools without imposing any changes and hence should be easier to adopt and implement.

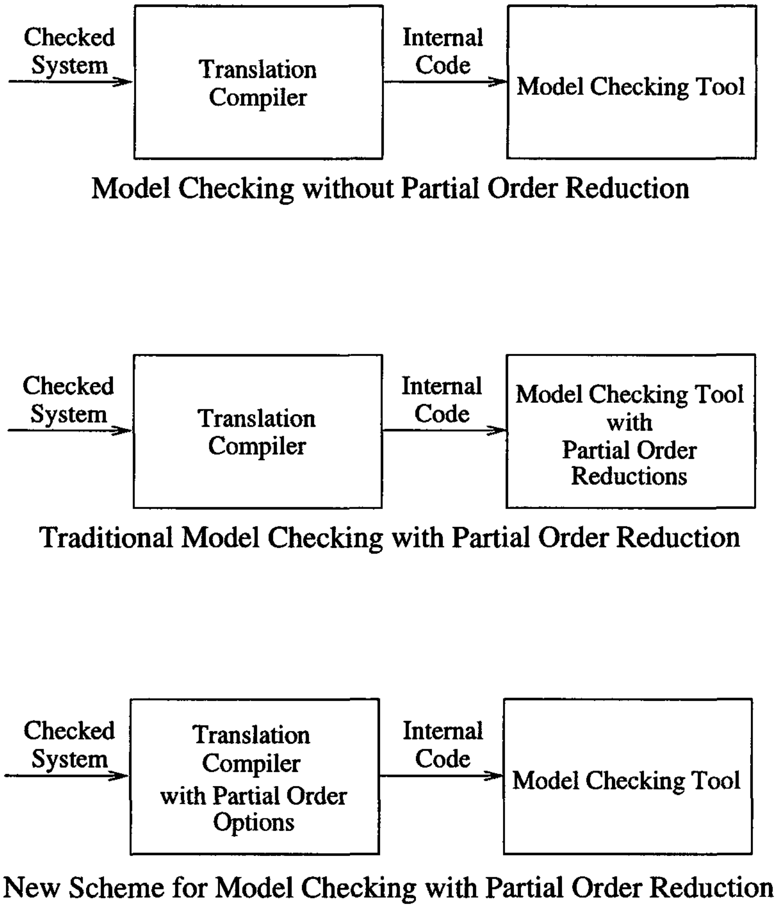
Checked System
Translation Compiler
Internal Code
Model Checking Tool
Model Checking without Partial Order Reduction
Checked System
Translation Compiler
Internal Code
Model Checking Tool with Partial Order Reductions
Traditional Model Checking with Partial Order Reduction
Checked System
Translation Compiler with Partial Order Options
Internal Code
Model Checking Tool
New Scheme for Model Checking with Partial Order Reduction

Fig. 1. Model Checking Scheme

2 A Simplified Search Algorithm

2.1 The Ample Sets Method

The reduction method we describe is applied to systems which are modelled as state-transition graphs. A state transition graph is defined as a tuple $M = (S, S_0, T, L)$, where S is the set of states, S_0 is the initial state set, T is a set of transitions $\alpha \subseteq S \times S$, and $L : S \rightarrow 2^{AP}$ a function that labels each state with some subset of a set AP of atomic propositions. A transition α is *enabled* in state s if there is some state s' for which $\alpha(s, s')$ holds. We denote the set of transitions enabled in s by *enabled*(s). If for any state s there is at most one state s' with $\alpha(s, s')$, we say that α is *deterministic* and we will write $s' = \alpha(s)$. In the following, we will consider only deterministic transitions. Note that although the transitions are deterministic (a usual practice in modeling concurrency), we can easily model non-deterministic choice (between different transitions that are enabled at the same time).

We introduce the key concept of *independent* transitions. These are transitions whose respective effects are the same, irrespective of their relative order.

Definition 1. Two transitions α and β are *independent* if for every state s the following two conditions hold:
Enabledness: If $\alpha, \beta \in$ *enabled*(s) then $\beta \in$ *enabled*$(\alpha(s))$ and $\alpha \in$ *enabled*$(\beta(s))$
Commutativity: If $\alpha, \beta \in$ *enabled*(s) then $\alpha(\beta(s)) = \beta(\alpha(s))$.

In other words, a pair of transitions is independent, if at any state executing either of them does not disable the other, and executing both in either order leads to the same state. Two transitions are called dependent if they are not independent. ·

To construct the reachable state space, model checking algorithms perform a traversal of the state-transition graph (typically depth-first or breadth-first search). The traversal starts from the set of initial states and successively constructs new states by exploring the transitions that are enabled in the current state. Partial order reduction differs from full state exploration in that at each step it considers only a subset of the transitions enabled at the current state s. This set is denoted by *ample(s)*. With a good choice of *ample(s)*, only a small fraction of the reachable state space will be explored. On the other hand, a number of conditions must be enforced on this set to ensure that the truth value of the checked property is preserved in the reduced model. In the following, we give a set of such conditions together with an informal explanation of their role. A complete treatment of these conditions together with a formal proof is given in [12].

Condition **C0** is the simplest and guarantees that if a state has a successor in the original model, it also has a successor in the reduced model.

C0 [Non-emptiness condition] *ample*$(s) = \emptyset$ if and only if *enabled*$(s) = \emptyset$.

C1 [Ample decomposition] On any path starting from state s, all the transitions appearing before a transition in *ample*(s) is executed, are independent of all the transitions in *ample*(s).

To explain **C1**, note that not every transition sequence in the original model may appear in the reduced model, since the latter is restricted to transitions from *ample*(s) at each state s. However, **C1** ensures that some transition from *ample*(s) may be taken in the reduced model without disabling any of the transitions in the original sequence. Consider any transition sequence σ starting in some state s_0. There are two possible cases:

- σ contains a transition $\alpha \in$ *ample*(s_0), in which case it is of the form $\beta_0\beta_1 \ldots \beta_n\alpha \ldots$. Condition **C1** then implies that α is independent of any $\beta_i, i \leq n$ and commutes with every one of these transitions. Then α can be taken in state s_0 and leaves the sequence $\beta_0\beta_1 \ldots \beta_n \ldots$ still enabled thereafter,
- the sequence does not contain any transitions in *ample*(s_0). Then an arbitrary transition $\alpha \in$ *ample*(s_0) can be taken in s_0, it is independent of all transitions in σ and therefore σ is still enabled in $\alpha(s_0)$ in the original model.

In order to be able to use the reduced model instead of the original one in verification, we also need to ensure that the checked property is not sensitive to the paths and states that have been eliminated from the reduced model. We consider as specifications next–time free linear time temporal logic (LTL without the next–time operator) formulas over the set of atomic propositions AP that label the states of the system. We call two paths *stuttering equivalent* if they are identical from the point of view of state labeling, after finite subsequences of successive states with the same labels have been collapsed into one state in each of them. It can be shown [11] that if two state transition graphs have the property that for any infinite path starting from an initial state in one of them there exists a stuttering equivalent infinite path in the other and vice versa, the two models satisfy the same set of next-time free LTL formulas.

We call a transition *invisible* if its execution has no effect on the state labeling. In other words, $\alpha \in T$ is invisible if $\forall s, s', \alpha(s, s') \Rightarrow L(s) = L(s')$. A state s is called *fully expanded* if *ample*(s) = *enabled*(s). In this case, all transitions are selected for exploration and no reduction is performed at this point.

C2 [Non-visibility condition] If there exists a visible transition in *ample*(s) then s is fully expanded.

Revisiting the two cases discussed for condition **C1** it can be seen that in each of these cases, α is an invisible transition (since s_0 is not fully expanded), and therefore the two paths considered will be stuttering equivalent.

Finally, we have to ensure that an enabled transition which does not belong to an ample set will eventually be taken. Otherwise, the constructions outlined in the discussion of **C1** may close a cycle in the reduced state graph while never taking a non-ample transition which is enabled throughout the cycle. Consequently some transitions can be ignored and the truth value of a specification in the two models may no longer be the same. Condition **C3** is introduced to eliminate this problem:

C3 [Cycle closing condition] At least one state along each cycle of the reduced state graph is fully expanded.[3]

As stated in the introduction, our goal was to develop a reduction algorithm which is not restricted to depth-first explicit state search, like the typical one described for instance in [9]. The principles and implementation of this algorithm are described below.

2.2 A Generic Partial Order Reduction

The cycle closing condition **C3** is very natural to check while performing a depth-first search. However, it cannot be checked directly when performing a breadth-first search (which is intrinsic to the symbolic methods), and therefore it seems that significant modifications to the model checking algorithms are needed to accommodate it (*cf* [1]). We show, however, that it is possible to ensure **C3** by performing static checks on the local state-transition graphs of each process. Conceptually, this method is able to perform a reduction (in terms of the number of reached states) at least as good as the traditional dynamic algorithms, although in practice there is a trade-off between the computational cost of the static reduction and computational savings afforded by the reduced model during the dynamic state space search. In fact, the most efficient balance in our algorithm may be achieved with varying degrees of state space reduction.

To describe our algorithm, we first note that both **C2** and **C3** limit the extent to which reduction can be performed: they define cases where a state has to be fully expanded. Moreover, if a cycle contains a visible transition, then **C2** guarantees that the state at which that transition is taken is fully expanded, and therefore **C3** holds for that cycle as well. This suggests that **C2** and **C3** can be combined into a single condition **C2'**:

C2' There exists a set of transitions T which includes all visible transitions, such that any cycle in the reduced state space contains a transition from T. When *ample*(s) includes a transition from T, s is fully expanded.

We call the set of transitions T ***sticky transitions***, since intuitively, they stick to all other enabled transitions.

To perform reduction during compilation of the modelled system, our goal is to determine a set T of sticky transitions that breaks all cycles of the reduced state graph, in order to guarantee **C2'**. We assume that the system to be verified is given as a set of component processes. Then, an easy way to find such a set T to look at the static control flow graph of each process of the system. Any cycle in the global state space projects to a cycle (or possibly a self-loop) in each component process. By breaking each local cycle, we are guaranteed to break each global cycle.

This suggests strengthening **C2'** to the following condition **C2"**:

[3] There are other stronger and weaker conditions that can be used instead of Condition **C3**. This particular version fits well with our framework.

C2'' There is a set of sticky transitions that include all visible transitions. Each cycle in the static control flow of a process of the modelled system contains at least one sticky transition, and if $ample(s)$ includes a sticky transition, then s is fully expanded.

An ideal algorithm would find a minimal set of sticky transitions, in order to maximize the possible reduction. However, this problem is at least as hard as reachability analysis. On the other hand, efficient reduction can still be achieved even without a minimal set. During the state search, priority is given to non-sticky transitions. In this way, full expansion of a state is avoided as much as possible, although eventually no cycle can be closed without performing one full expansion. It is possible therefore that several sticky transitions are delayed until all of them can be taken from the same state, which reduces the effect of selecting too many sticky transitions.

Even with delaying sticky transitions, it is still important that the static analysis generates a small number of sticky transitions, and yet is simple enough not to require excessive overhead. The next section presents such an algorithm which is heuristically likely to generate a smaller number of sticky transitions than required by **C2''**. The set of sticky transitions found by the algorithm guarantees **C2'** and in the worst case it corresponds to **C2''**.

2.3 Finding Sticky Transitions

We assume that the system to be analyzed is given as a set of variables V and a set of processes $\{P_1, P_2, \ldots, P_N\}$. We also assume that, for each process P_i, there exists a variable $cp_i \in V$ called the *control point* (or *program counter*) of process P_i, which always keeps the current local state of the process. A transition of P_i updates cp_i (not necessarily changes its value) and also updates some other variables from V. The state space of the system is simply given by all possible valuations of the variables in V. The state-transition graph of the system is derived from the local state-transition graphs of the processes by using interleaving semantics to model concurrency. A *local* (resp. *global*) cycle is a cycle in the state-transition graph of a process (resp. the system). An *execution* of a cycle is the execution of all the transitions in the cycle starting from a state in the cycle.

An execution of a local cycle of a process P_i restores the value of cp_i. But along the cycle, the values of variables other than cp_i can be changed as well, without necessarily being restored by a complete execution. We call this the side effect of a local cycle on a variable and observe four different types of side effects: (1) *decrementing effect*, if the execution of the cycle always reduces the value of the variable, (2) *incrementing effect* if the execution of the cycle always increases the value of the variable, (3) *complex effect* if the effect of the execution of the cycle on the variable cannot be determined statically, and (4) *no effect* if the variable is not changed by any of the transitions in the cycle.

If the side effect of a local cycle c is incrementing or decrementing over the value of a variable v, it is impossible to have a global cycle in which only c is executed. There must be some other local cycle c' executed in the global cycle to compensate for the side effect of c on v. For every global cycle in which c is executed, c' must be executed as well. Therefore, there is no need to select a sticky transition from both c and c' since neither c nor c' can appear alone in a global cycle.

Let C denote the set of local cycles in the system. We assume the existence of a function $f : C \times V \longrightarrow \{-,+,\star,0\}$ such that for $c \in C$ and $v \in V$, $f(c,v) = -$ ($f(c,v) = +$, $f(c,v) = \star$, $f(c,v) = 0$, respectively) means a decrementing effect (incrementing, complex, no effect, respectively) on v by c. One can always assume $f(c,v) = \star$ if v is updated within c but the side effect is difficult to analyze.

Definition 2. A set of local cycles $H \subseteq C$ *covers* another set of local cycles $G \subseteq C$ if any global cycle that contains (projects to) a local cycle $c \in G$ also has to contain some local cycle $c' \in H$.

In the particular case where G is a singleton set $\{c\}$, we will simply say that H covers c.

We can effectively find a set of cycles that covers a local cycle c by considering the effect of c on some variable v. For a given local cycle c and a variable v, let c_v be the set of local cycles that can compensate the incrementing or decrementing effect of c on v which is formally defined as:

$$\begin{aligned} c_v = \{c' \in C \mid\ & (f(c,v) = - \textit{ and } f(c',v) \in \{+,\star\}) \text{ or} \\ & (f(c,v) = + \textit{ and } f(c',v) \in \{-,\star\})\}. \end{aligned}$$

Since c_v contains *all* cycles that can have the opposite effect on v compared to c, it follows that c_v covers c. This implies that if for some variable v, all cycles in c_v have a sticky transition, there is no need for c to have a sticky transition.

Our goal is to find a subset T of sticky transitions that breaks (when removed from the local process graphs) some set H of local cycles such that H covers the entire set of local cycles C. Then, since every global cycle contains some local cycle $c \in C$, it also has to contain a cycle from H, and with it a sticky transition. Consequently, condition **C2'** holds.

To find such a set, note that trivially H covers H for any $H \subseteq C$. We also have the following lemma:

Lemma 3. *Let $H, G \subseteq C$ and $c \in C$. If H covers G and G covers c, then H covers $G \cup \{c\}$.*

Proof: We need to show that for any global cycle C_1, if C_1 contains a local cycle $g \in G \cup \{c\}$ then it has to contain a local cycle in H. If $g \in G \cup \{c\}$ then we have two cases, either $g \in G$ or $g = c$.
Case (i): If $g \in G$, then since H covers G, C_1 must have a local cycle in H.
Case (ii): If $g = c$, then C_1 has to contain a local cycle $g' \in G$ since G covers c. Furthermore, since C_1 contains $g' \in G$ and H covers G, C_1 contains some cycle in H.
Together, these two cases show that H covers $G \cup \{c\}$.

The Algorithm 1 given in Fig. 2 uses this lemma to compute a set H such that H covers C. It alternates between analyzing the effect of local cycles on variables to increase the covered set G and adding cycles to H if there are still uncovered cycles in C.

Algorithm 1

```
0. choose $H \subseteq C$, let $G := H$
1. loop
2.   do
3.     let $updated := false$
4.     $\forall c \in C \setminus G, \forall v \in V$
5.       if $f(c, v) \in \{-, +\}$ and $c_v \subseteq G$ then
6.         let $G := G \cup \{c\}$
7.         let $updated := true$
8.   while $(updated)$
9.   if $(G = C)$ return $H$
10.  let $H := H \cup C_{add}$, $G := G \cup C_{add}$ for some $C_{add} \subseteq C \setminus G$, $C_{add} \neq \emptyset$
11. endloop
```

Fig. 2. An algorithm to find $H \subseteq C$ such that H *covers* C

It is possible not to take a variable v into account during the local cycle analysis by simply assuming that $f(c, v) = \star$ for all local cycles. One can also assume the existence of auxiliary variables to produce the dependency relation between cycles. For example, if there is a variable q of type queue in the system, we can assume that there is also an integer variable q_l, which always keeps the number of elements in this queue variable. It is hard to define the side effect of a push or pop operation on q but they are incrementing and decrementing on q_l respectively. In the extreme case where $\forall c \in C, \forall v \in V$, $f(c, v) \in \{\star, 0\}$, Algorithm 1 terminates with $H = C$ as the worst case which corresponds to satisfying Condition **C2"** since we have to chose a sticky transition from each local cycle.

The selection of initial set of marked cycles, let's call it C_m, can be arbitrary. A good starting value is given by the sticky transitions which are already required by **C2'**. In particular, C_m can be chosen to be the set of all cycles that include a visible transition.

2.4 The COSPAN Implementation

The static partial order reduction technique explained in this paper has been implemented for SDL and S/R source–target pair of languages. Nevertheless, the method is not specific to this pair of languages. We give the details of this particular implementation in this section.

Our method of applying the reduction entails the modification of the analyzed system such that a transition which is not in the ample set for a given state, is simply not enabled. In other words, the set of enabled transitions at some state in the modified system is exactly an ample set at that state if the original system were analyzed with a modified search algorithm. This property enables us to use any search technique to analyze the modified system. In our case, we are able to use either explicit and symbolic search techniques and also apply localization reduction [10] together with the partial order reduction.

In order to achieve in COSPAN a partial order reduction that is independent of the search control, we exploit the selection mechanism of S/R. The language provides *selection variables*, which are not part of the state, and thus do not incur any memory overhead. When deciding on the successor state, each process chooses non-deterministically among some possible values of its selection variables. The choice of any process can be dependent on the choice of the selections of the other processes (as long as this relationship is acyclic).

In the compilation phase from SDL to S/R, first the visible transitions are tagged as sticky. Algorithm 1 is then executed to find a sufficient set of sticky transitions with the initial selection C_m being the set of local cycles that include a visible transition. Also for each local state of a process, we calculate whether the transitions departing from that local state satisfy Condition **C1**. If the process has only internal transitions (the transitions in which only the local variables are referred), then it is clear that the transitions originating from that local state of the process satisfy **C1** since no other process can refer to those variables. Similarly, when the process has only enabled receiving transitions, the transitions of the process again satisfy **C1**. Although the send transition of another process can change the same message queue from which the receiving transition reads, their execution order does not matter. Depending on the topology of the system, even a send transition of a process can also satisfy **C1**, for example if there is no other process that can send a signal to the same message queue. Note that, the compilation is dependent on the property to be checked (or more precisely, on the set of visible transitions). Therefore, a new compilation is required for each property that impose different visible transitions in the system.

In the current version of our compiler, a process is considered to be ample at a state if it does not have a sticky transition and all of its transitions satisfy **C1** at its current local state. Each process sets a global combinational flag to true or false depending on its ampleness at a global state. From all the ample processes at a state, a process with the least number of outgoing transitions is chosen as the candidate for execution. If more than one process has the least number of transitions, a static priority (index number) is used to chose only one process. If there is no ample process at a state then all the processes are chosen as candidates for execution. A process does not have any enabled transition unless it is selected as one of the candidates for execution. This candidate election mechanism is implemented using the primitives of S/R and is embedded in the source code of the analyzed system without causing any state space overhead.

We added approximately 1000 lines of code to the original compiler (which was around 9000 lines before the addition) to implement the reduction.

3 Experimental Results

This section gives experimental results for our method. The examples specified in SDL are translated into S/R using the compiler incorporating the static partial order reduction approach explained in this paper.

The first example is a concurrent sort algorithm. There are $N + 1$ processes which sort N randomly generated numbers. One of the processes simply generates N random numbers and sends them to the next process on the right. Each process that receives a new number compares it with the current number it has and sends the greater one to the process on the right. The rightmost process receives only one number which is the largest one generated by the leftmost process. The second example is a leader election protocol given in [5]. It contains N processes, each with an index number, that form a ring structure. Each process can only send a signal to the process on its right and can receive a signal from the process on its left. The aim of the protocol is to find the largest index number in the ring. The protocol is verified with respect to all possible initial states. The final example is an asynchronous tree arbiter as taken from [4] whose purpose is to solve the mutual exclusion problem. A resource is arbitrated between N users by a tree of arbiter cells. Each arbiter cell can have at most two children and forwards a request coming from its children to the upper level of the tree. When an arbiter cell receives the grant, it passes the grant to the child that requested the resource. If both of the children are requesting, the grant signal is sent nondeterministically to one of them. When the resource is released, the release information is sent to the root of the tree along the branch connecting the root and the user that released the resource. An acknowledgement is also sent back by the root to the user, using the same branch in the tree. Table 1 gives the measurements we have obtained so far on these examples.

The examples above showed that in case of small state spaces, the symbolic search with partial order reduction is more expensive than an explicit search with partial order reduction. It is even more expensive than a symbolic search on original system without any partial order reduction. As the state space gets bigger, the symbolic search with partial order reduction, starts doing better than the symbolic search without reduction. For large systems, the symbolic search with partial order reduction becomes the fastest of all the alternatives.

The concurrent sort example has an interesting property for the application of Algorithm 1. We have introduced an artificial integer variable for each message queue in the system that is assumed to keep the number of messages in the queue. When Algorithm 1 is executed by taking into account only these artificial variables with $C_m = \emptyset$ initially, it returns $H = \emptyset$. The reason of this is that, even though there are cycles in the local graphs of the processes, the global state space has no cycles and this can be determined by a syntactic analysis.

Since the ample set reduction is applied completely statically, it cannot benefit from all the information available to a dynamic algorithm. For example,

Experiments	No Reduction (*no of states*)	Ample Reduction (*no of states*)
Sort with $N = 2$	191	66
Sort with $N = 3$	4903	553
Sort with $N = 4$	135329	4163
Sort with $N = 5$	3940720	29541
Leader with $N = 2$	383	107
Leader with $N = 3$	11068	490
Leader with $N = 4$	537897	3021
Leader with $N = 5$	26523000	21856
Arbiter with $N = 2$	73	48
Arbiter with $N = 4$	18247	4916
Arbiter with $N = 6$	3272700	358352

Table 1. Experimental Results

Condition **C3** is satisfied by predicting the cycles in the global state space at syntactic level. It is possible that Algorithm 1 will try to break global cycles that can actually never occur. A reduction algorithm that breaks global cycles as they appear during the analysis seems to be more fine tuned for the reduction. However, the produced experimental results are as good as those obtained by dynamic algorithms.

4 Summary

Model checking tools are highly complex and required to have a a good performance. On the other hand, the state space explosion problem forces the tool implementors to incorporate the possible reduction techniques into the tools, making the implementation more complex. Frequently, it is not straightforward to implement a reduction technique on top of the search technique used by a model checker. Until recently [1], there were no implementations that combine partial order reduction and symbolic search techniques although both methods were known for a long time and had good implementations separately.

We have demonstrated a way to compute a partial order reduction of an asynchronous system statically. This facilitates implementation of the reduction into model-checking tools without the need to alter the search algorithms. In particular, our method allows combining partial order reduction with symbolic search. Although our implementation of the method uses SDL and S/R as the source and the target languages, the method itself is not specific to these languages.

Experimental results indicate that for small models, static partial order reduction is faster with an explicit state representation. However, for large models, the symbolic search is not only faster, but completes on models which are computationally infeasible with reduction based on an explicit state search.

References

1. R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Partial order reduction in symbolic state space exploration. In *Proceedings of the Conference on Computer Aided Verification (CAV'97)*, Haifa, Israel, June 1997.
2. E. Bounimova, V. Levin, O. Başbuğoğlu, and K. Inan. A verification engine for SDL specification of communication protocols. In S. Bilgen, U. Çağlayan, and C. Ersoy, editors, *Proceedings of the First Symposium on Computer Networks*, pages 16–25, Istanbul, Turkey, May 1996.
3. C.T. Chou and D. Peled. Formal verification of a partial–order reduction technique for model checking. In *Proceedings of the Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 241–257, Passau, Germany, 1996. Springer–Verlag. Volume 1055 of Lecture Notes in Computer Science.
4. D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, 1989.
5. D. Dolev, M. Klave, and M. Rodeh. An $O(n\log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3:245–260, 1982.
6. 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.
7. R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In *Proc. CAV'96*, volume 1102, pages 423–427. LNCS, 1996.
8. G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1992.
9. G.J. Holzmann and D. Peled. An improvement in formal verification. In *Formal Description Techniques 1994*, pages 197–211, Bern, Switzerland, 1994. Chapman&Hall.
10. R. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, 1994.
11. L. Lamport. What good is temporal logic. In *IFIP Congress*, pages 657–668. North Holland, 1983. in Computer Science 115.
12. D. Peled. Combining partial order reductions with on-the-fly model checking. *Formal Methods in System Design*, 8:39–64, 1996.
13. *Functional Specification and Description Language (SDL), CCITT Blue Book, Recommendation Z.100.* Geneva, 1992.
14. 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.