Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science 128 (2005) 63–74
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Distributed Partial Order Reduction of State Spaces[1]

L. Brim, I. Černá, P. Moravec, J. Šimša

Faculty of Informatics, Masaryk University, Brno, Czech Republic

Abstract

State space explosion is a fundamental obstacle in formal verification of concurrent systems. Several techniques for combating this problem have emerged in the past few years, among which the two we are interested in are: partial order reduction and distributed memory state exploration. While the first one tries to reduce the problem to a smaller one, the other one tries to extend the computational power to solve the same problem. In this paper, we consider a combination of these two approaches and propose a distributed memory algorithm for partial order reduction.

Keywords: Distributed model-checking, partial order reduction, LTL model-checking

1 Introduction

Concurrent systems are composed of systems that can cooperate concurrently and communicate with each other. Concurrent systems typically exhibit an extremely large number of different behaviors due to the combinatorial explosion resulting from all possible interactions between the components and many possible race conditions that may arise between them. Model checking based on state space exploration is a common technique for determining that all possible behaviors of the system are compatible with the given property. It consists of exploring the state graph (*state space*) representing the combined behavior of all system components. For finite state systems the graph can be explored exhaustively. The main limit of the technique is the size of the state

[1] Research supported by the Grant Agency of Czech Republic grant No. 201/03/0509
[2] Email: {`brim,cerna,xmoravec,xsimsa`}`@fi.muni.cz`

doi:10.1016/j.entcs.2004.10.019

graph that can grow exponentially with the size of the system description (*state space explosion*).

Several methods are used to overcome the state space explosion. In this paper we make a delve into a combination of two of them: the partial order reduction method and distributed methods.

The partial order reduction method is aimed at reducing the size of the state space that needs to be searched. It exploits the fact that exploring all interleavings of concurrent events might not be necessary for checking the property and as such it is best suited for asynchronous systems. The method consists of constructing a reduced state space; the full state space, which may be too big to fit into a memory, is never constructed. This speeds up the construction of the state space, uses less memory, and results in a more efficient model checking. The behaviors exemplified with the reduced state space form a subset of the behaviors of the full state space, however, their stuttering invariant properties remain valid. Informally, stuttering invariance means that the truth value of a property on an infinite sequence of states does not change if states in the sequence are repeated a finite number of times. We consider properties expressed as formulas of the linear time temporal logic LTL. A simple way to restrict properties that can be expressed in LTL to stuttering invariant properties is to disallow the use of the *next* operator (LTL_{-X}). There are several accomplishments of the method, for more details see [12,19,21]. The approach our algorithm is based on is summarized in Section 2.

Distributed methods cope with the state explosion by distributing the state space among several workstations in a network with the aim to increase the computational power (especially random access memory) by building a powerful parallel computer as a network (cluster) of workstations. The workstations communicate through a message passing interface and in mutual cooperation explore the whole state space. There is an extensive interest in building distributed verification tools (e.g. [1,2,3,5,6,10,22]). In [4,13,14] a combination of symbolic model checking and distribution has been considered.

A natural question is how to combine the partial order method with enumerative distribution allowing thus construction of a reduced state space in a distributed setting. Two approaches combining the methods have been presented in [15] and [17]. The method we propose in the paper can be understood as a generalization of both of them – for a detailed comparison see Section 5. The basic idea behind the parallelization is in dividing the generation of the reduced state space into independent subtasks that can be performed in an arbitrary order in parallel. This is achieved by splitting the search stack into parts determined by fully expanded states. The method is described in Section 3. Section 4 summarizes the experiments we have performed.

2 Partial Order Reduction Method

In this section we give a brief review of the partial order reduction method following mainly the presentation of [9]. The concurrent systems that we analyze are modeled as state transition systems (labeled transition systems). 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 system* is then defined as a tuple $M = (S, s_0, \Delta, L)$, where $s_0 \in S$ is an *initial state*, Δ 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 $\alpha \in \Delta$ is *enabled* in a state s iff there is a state s' such that $\alpha(s, s')$. The set of all transitions enabled in a state s is denoted *enabled(s)*. We presuppose that transitions are deterministic, i.e., for every α and s there is at most one s' with $\alpha(s, s')$, and denote it as $\alpha(s) = s'$. If $\alpha(s, s')$ we say that s' is a *successor* of s.

As has been mentioned in the introduction, the partial order method exploits transitions that can be executed concurrently and interleaved in either order. This can be formalized by defining an independence relation on pairs of transitions that can execute concurrently.

Definition 2.1 An *independence* relation $I \subseteq \Delta \times \Delta$ is a symmetric, antireflexive relation, satisfying the following two conditions for each state $s \in S$ and for each $(\alpha, \beta) \in I$:

(i) Enabledness – If $\alpha, \beta \in \mathit{enabled}(s)$ then $\alpha \in \mathit{enabled}(\beta(s))$.

(ii) Commutativity – If $\alpha, \beta \in \mathit{enabled}(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$.

The *dependency* relation is the complement of I.

The problem to find out the smallest possible dependency relation of a given model of an asynchronous system is as hard as the reachability problem. Therefore, heuristic methods are utilized for an efficient computation of a dependence relation according to the conditions mentioned above.

The independence relation suggests a potential reduction to the state transition system by selecting only one from the independent transitions originating from a state s. However, this cannot guarantee that the reduced state transition system is a correct replacement of the full one as it does not take into account the property to be checked. Also, eliminating one of the intermediate states $\alpha(s)$ or $\beta(s)$ 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.

First, we make it precise what it means that a property is taken into account by defining the concept of *visibility* of a transition.

Definition 2.2 A transition $\alpha \in \Delta$ is *invisible* with respect to a set of propositions $AP' \subseteq AP$ if for each pair of states $s, s' \in S$ such that $\alpha(s, s')$, $L(s) \cap AP' = L(s') \cap AP'$ holds. A transition is *visible* if it is not invisible.

The set AP' is usually induced by the set of atomic propositions included in the verified formula.

The reduced state transition system, denoted by M_R, is generated by a modified generation algorithm, which explores only a subset of transitions, called an *ample set*, enabled at each state encountered during the generation. The ample set can be defined in a manner that does not depend on the particular way the state transition system is generated by a set of *conditions* relating the full state transition system and the corresponding reduced one. Note, that there could be more than one ample set satisfying the conditions for a given state. We say that a state s is *fully expanded* whenever $\mathit{ample}(s) = \mathit{enabled}(s)$.

Definition 2.3 Let AP' be a set of atomic propositions. *Ample conditions* with respect to the set AP' are:

C0 $\mathit{ample}(s) = \emptyset$ iff $\mathit{enabled}(s) = \emptyset$.

C1 Along every path in the full state graph M that starts at s, the following condition holds: a transition that is dependent on a transition in $\mathit{ample}(s)$ cannot be executed without a transition in $\mathit{ample}(s)$ occurring first.

C2 If $\mathit{enabled}(s) \neq \mathit{ample}(s)$, then every $\alpha \in \mathit{ample}(s)$ is invisible w.r.t. to AP'.

C3 (*cycle closing condition*) A cycle in the reduced state graph M_R is not allowed if it contains a state in which some transition α is enabled, but is never included in $\mathit{ample}(s)$ for any state s on the cycle.

The conditions characterize the ample sets needed to generate reduced state transition systems sufficient for checking safety and liveness properties. In particular, the resulting reduced state transition system generated is guaranteed to be stuttering equivalent to the full system and consequently all LTL$_{-X}$ properties are preserved.

Theorem 2.4 *Let $M = (S, s_0, \Delta, L)$ be a state transition system and $M_R = (S_R, s_0, \Delta_R, L)$ be a reduced state transition system satisfying the ample conditions **C0, C1, C2** and **C3** with respect to a set of atomic propositions AP'. Let φ be a formula of LTL$_{-X}$ over AP'. Then φ is satisfied in M if and only if it is satisfied in M_R .*

Thus the problem whether a given LTL$_{-X}$ formula φ is satisfied in a given system M can be reduced to the problem whether φ is satisfied in the reduced system M_R.

3 Distributed Computing of the Reduced State Space

For the distributed computation we assume a network of collaborating *nodes* (workstations, computers) with no global memory. Communication among the nodes is realized by sending messages only. In the distributed computation the state transition system is divided into parts, one part per each node.

Our aim is to design a distributed memory algorithm for computing the reduced state transition system. The reduced system is computed by a generation algorithm which systematically explores states in such a way that for every state s it chooses a set $\mathit{ample}(s) \subseteq \mathit{enabled}(s)$ and follows the transitions from $\mathit{ample}(s)$ only. The key part of such an algorithm is without any doubts the distributed checking of the ample conditions.

While checking conditions **C0** and **C2** is easy and can be done locally, checking conditions **C1** and **C3** is as hard as solving the reachability problem. The condition **C1** can be checked locally using the same approximating heuristics as in the sequential case (see [9]). The cycle closing condition **C3** is the only one which is difficult to be checked in a distributed environment. In the sequential case when exploring the state graph using *depth first search*, the condition **C3** is checked *in constant time* using the *search stack*. In fact, the following stronger condition is used instead of **C3**.

C3′ If a state s is not fully expanded, then no transition in $\mathit{ample}(s)$ leads to a state on the search stack.

Our aim is to develop a counter part of the condition **C3′** for the depth first search based generation of the state transition system which is distributed among several nodes. To check the original condition **C3′** in the distributed setting with the same effectiveness is extremely expensive. Therefore, we propose to use a more suitable condition. The new condition is motivated by the observation that during the depth first search only a part of the search stack is needed in order to ensure the condition **C3′**. In particular, the significant part of the search stack is the one between the top of the stack and the topmost state that has been fully expanded. This is because after a state has been fully expanded all the cycles reaching this state through the search stack contain this state as a fully expanded one. Based on this simple observation we can split the reduction (generation) process into independent subtasks. Each time a state is fully expanded, we start a new search with an empty search stack. This is particularly suitable for distribution as we do not need to care about transferring search stacks among the nodes. Several subtasks can be performed in parallel on different nodes. To deal with "global cycles" (stretching over more than one node), we fully expand a state whenever crossing to a different node. To sum up, we use the following cycle closing condition.

C3″ If a state s is not fully expanded, then no transition in $ample(s)$ leads to a state on the local search stack nor to a state owned by a different node.

As mentioned above, we assume that the state transition system (actually the states only) is partitioned over several *nodes*. The partition function is denoted $owner()$. The owner of the initial state s_0 is denoted $manager$.

The main idea of the distributed algorithm is the following. Each node maintains a set *waiting* of states from which the generation of the reduced state transition system is to be started. A *manager* initiates the entire computation by starting the first depth first search procedure from the initial state. Whenever a new state s is visited a set $ample(s) \subseteq enabled(s)$ of transitions is computed. We always try to select a set that fulfills the ample conditions; in particular in case of **C3″** it does not include a transition leading to a search stack nor to another node. If we do not succeed, the current state is fully expanded. There are two possible scenarios.

In the case the state s is fully expanded, every successor s' of the state s is inserted into the set *waiting*. If $owner(s')$ differs from $owner(s)$ a message is sent to the owner of s' to do so. The depth first search then backtracks from the state s.

Otherwise, the depth first search continues in the state transition system generation following transitions from $ample(s)$ only.

After the depth first search ends, all incoming messages are processed. Then a state from the set *waiting* is picked and a new depth first search is initiated from it. This step is repeated until the set *waiting* is empty. Once the set *waiting* is empty and there are no incoming messages, the node starts to idle. If all nodes idle and there are no pending messages the algorithm terminates.

Note that a state s is fully expanded whenever there is no $ample(s)$ such that no transition from $ample(s)$ points to a different node. This is not really necessary in general. The main reason for the full expansion when crossing to another node is to deal with "global" cycles. However, such a cycle has to pass through the same node at least twice. In the following we describe two simple heuristics to decrease the number of full expansions made while ensuring the satisfaction of the condition **C3′**.

The first heuristic employs an ordering on the involved nodes. If it is not possible to select an $ample(s)$ such that no transition from $ample(s)$ points to a different node which is strictly greater than $owner(s)$ the state is fully expanded.

The second heuristic is more involved and can give a better reduction. It keeps track of visited nodes during each depth first search. To that end we associate with each depth first search an array *history* of boolean values. The length of the array is fixed and equals to the number of nodes involved. The i-th element of the array *history* is true if and only if the search has visited some state owned by the node i. Values of the array *history* are taken into account when computing the set $ample(s)$ for a state s according to the condition:

C3‴ If a state s is not fully expanded, then no transition in $ample(s)$ leads to a state on the local search stack nor to a state owned by a different node such that the current search has already visited this node.

Again, there are two possible scenarios. In the case $ample(s)$ is a proper subset of $enabled(s)$ the depth first search continues in state transition system generation following the transitions from $ample(s)$ as follows:

- For each transition which points to a state s' such that $owner(s')$ differs from $owner(s)$ we create a copy $history'$ of the array *history* where we set $history'[owner(s)] = true$ and send a message containing the tuple (s', $history'$) to the owner of s'.
- The depth first search then continues from s considering only transitions which point to a state s' such that s and s' have the same owner. If the state s' has been visited and for its history $s'\_history$ and for all indexes i the implication $history[i] \Rightarrow s'\_history[i]$ is valid then the state is considered as visited. Otherwise the depth first search enters the state s' and s' is stored either with *history* (if not visited before) or with the history $history \vee s'\_history$.

The second scenario is $ample(s) = enabled(s)$, i.e., s is fully expanded. In this case every successor s' of the state s is inserted into the set *waiting* together with the array *empty_history* ($empty\_history[i] = false$ for every node i). If $owner(s')$ differs from $owner(s)$ a message is sent to the owner of s' to do so. The depth first search then backtracks from the state s.

The pseudo code of the algorithm (including the described "history" heuristic) is given in Figure 1. The function $visited(s, history)$ returns *true* if and only if the state s has been visited and the array *history* holds no new information about nodes visited by the search prior to the state s (for all indexes i the implication $history[i] \Rightarrow s\_history[i]$ is valid).

Proposition 3.1 *The algorithm terminates and the reduced state transition system satisfies conditions **C0**, **C1**, **C2**, and **C3**.*

The proposition follows from the above given arguments. The formal proof of correctness is technically quite involved and we skip it.

As regards the time and space complexity we evaluate the complexity with respect to an individual node. Let n be the number of states assigned to a node, *e_out* be the number of edges out coming from states assigned to a node,

```
proc Main() /* for node i */
  if $i =$ manager then $waiting.push(s_0, empty\_history)$; fi
  while $\neg finished$ do
        $process\_messages()$;
        while $waiting \neq \emptyset$ do
              $(state, history) := waiting.pop()$;
              if $\neg visited(state, history)$ then $Dfs(state, history)$; fi
              $process\_messages()$;
        od
  od
end

proc Dfs(state, history) /* for node i */
  if $ample(state) = enabled(state)$
    then
         foreach $t \in ample(state)$ do
           if $owner(t(state)) = i$
              then if $\neg visited(t(state), history)$
                      then $waiting.push(t(state), empty\_history)$; fi
               else $send(owner(t(state)), (t(state), empty\_history))$;
           fi
         od
     else
         foreach $t \in ample(state)$ do
           if $owner(t(state)) = i$
              then if $\neg visited(t(state), history)$
                      then $Dfs(state, history)$; fi
               else $send(owner(t(state)), (t(state), history'))$;
           fi
         od
  fi
end
```

Fig. 1. The Distributed Algorithm

and e_in be the number of edges such that their endpoints are states assigned to a node.

Proposition 3.2 *The time complexity of the computation without any heuristic performed on a node is $\mathcal{O}(n + e\_out + e\_in)$. The complexity of the computation with the "history" heuristic is $\mathcal{O}(P.(n + e\_out + e\_in))$ where P is the number of nodes participating in the reduced state transition system generation. The space complexity is $\mathcal{O}(n)$ respectively $\mathcal{O}(P.n)$.*

Proof. The computation without the heuristic explores every state and all its out coming edges exactly once. Moreover, the incoming messages have to be maintained which takes time $\mathcal{O}(e\_in)$. In the case the heuristic is employed a state and its out coming edges are explored every time its history has been changed. Since the history monotonically increases, the number of re-explorations is at most P.

In both version the number of visited states (with their *history*) determines the space complexity.

4 Experiments

We have implemented the distributed algorithm with the "history" heuristic as proposed in Section 3. The implementation has been done in C++ and the experiments have been performed on a cluster of thirteen Intel Pentium 4 2.6 GHz workstations with 1 GB of RAM each interconnected with a fast 100Mbps Ethernet. In the implementation the state transition system is partitioned among the workstations using random hash function as this guarantees an even distribution. Once the system is partitioned no re-balancing is performed.

We performed several tests in order to evaluate the reduction potential and scalability of the algorithm. We considered four groups of models. The first group consisted of models corresponding to the Peterson algorithm for the mutual exclusion problem parametrized by the number n of processes (denoted as $PA(n)$). The second group consisted of models corresponding to the token ring algorithm parametrized by the number n of processes ($TRA(n)$). The third group consisted of models corresponding to the alternating bit protocol parametrized by the number n of bits which can be lost in a row ($ABP(n)$). Finally, the fourth group consisted of models corresponding to the simple sender-receiver protocol parametrized by the number n of bits that can be lost in a row ($SRP(n)$). The reduction of state transition systems was done with respect to *atomic propositions* taken from LTL$_{-X}$ formulas $GF(P.cs)$ expressing that the process P will enter its critical section infinitely many times (for $PA(n)$ and $TRA(n)$) and $GF(Receiver.0 \vee Receiver.1)$ expressing that the receiver will receive some value infinitely many times (for $ABP(n)$ and $SRP(n)$).

Model	**Full**	**SPIN**	**Ratio**	**FDFS**	**Ratio**
PA(4)	2239099	1470588	65.7 %	1449397	64.8 %
TRA(15)	1474559	116	$7 \cdot 10^{-5}$ %	116	$7 \cdot 10^{-5}$ %
TRA(16)	3145727	124	$3.9 \cdot 10^{-5}$ %	124	$3.9 \cdot 10^{-5}$ %
ABP(11)	1965620	1226483	62.4 %	1073803	54.7 %
ABP(12)	2302468	1433301	62.3 %	1256987	54.6 %
SRP(26)	1687102	1332503	79.0 %	1246693	73.9 %
SRP(27)	1814110	1437949	79.3 %	1346365	74.2 %

Fig. 2. Reductions

The results of *sequential* experiments are presented in Figure 2. We compare the number of reachable states of the full state transition system (*Full*) with the size of the reduced state transition system. The reduced state transition system is generated using a depth first search procedure where a state

s is fully expanded whenever a transition from $ample(s)$ points to the search stack. This algorithm is widely used and as it is implemented in the SPIN model checker we denote it *SPIN*. Then the reduced state transition system is generated using our algorithm *FDFS* (an abbreviation of fragmented depth first search).

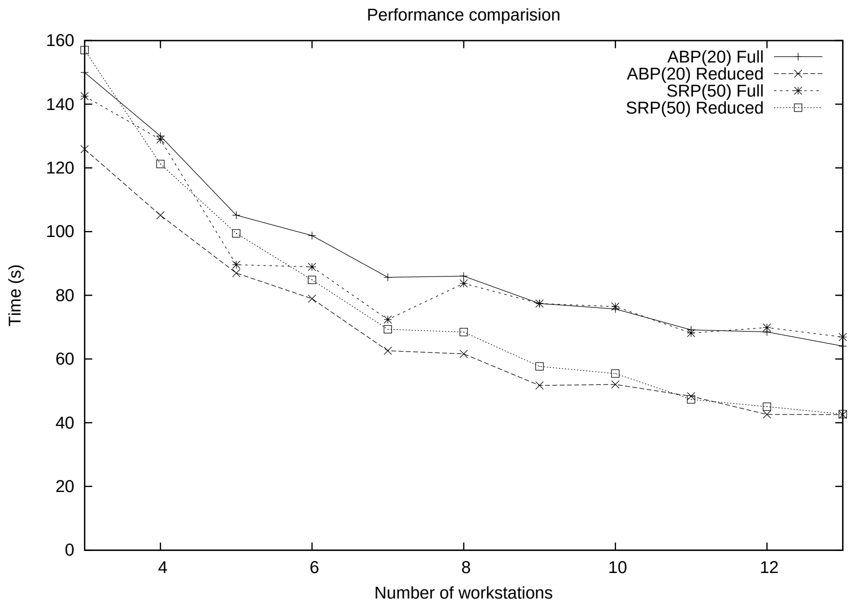

Fig. 3. Speedup in Distributed Environment

The actual performance of our algorithm with the "history" heuristic in the *distributed* environment is presented in Figure 3. Its performance is compared with a distributed algorithm generating the full state transition system. At least three workstations were needed in order to generate the full state transition system for both $ABP(20)$ and $SRP(50)$ (5954564 and 6007102 states respectively). The reduction ratio (54.5% and 73.8% respectively) was independent on the number of nodes and the figure shows only the dependence between the number of nodes and the time of computation.

The experiments demonstrate that the reduction ratio of our algorithm can really be better than that achieved by more conservative cycle closing condition. At the same time the algorithm in the distributed environement scales well and requires less time than the distributed reachability algorithm.

5 Related Work and Conclusions

The first attempt to combine the methods is by Lerda and Sisto ([15]) who augment the distributed SPIN model checker with partial order reduction. The algorithm allows only for reachability checking and uses the conservative cycle closing condition (successors that are hold outside the node where they are computed are always assumed to be currently in the search stack).

In [17,18] the distributed algorithm *Twophase* is proposed. In contrast to SPIN's algorithm, *Twophase* is much simpler as it works only with singleton ample sets, i.e., whenever there is no singleton satisfying the ample conditions, the state is fully expanded. While generating singleton ample sets the computation does not cross to a different node and the cycle closing condition thus can be checked locally. In a parallel context the algorithm realizes just the reachability checking.

In comparison to these two approaches the algorithm we have proposed can increase the reduction by considering not only singleton ample sets and by using less conservative cycle closing condition. At the same time the algorithm for generating the reduced state space can be easily combined with distributed LTL model checking algorithms [7,8,1] when the reduced state space is generated and at the same time checked for correctness with respect to a given LTL property. However, up to now we experimentally tested our algorithm only for scalability and reduction effectiveness and testing its effectiveness in the full LTL model checking is a future work.

References

[1] J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model-checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 106–115. IEEE Computer Society, Oct. 2003.

[2] G. Behrmann, T. S. Hune, and F. W. Vaandrager. Distributed timed model checking — how the search order matters. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 216–231. Springer-Verlag, 2000.

[3] A. Bell and B. R. Haverkort. Sequential and distributed model checking of petri net specifications. In *Proceedings of the 1st International Workshop on Parallel and Distributed Model-Checking (PDMC'02)*, volume 68(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

[4] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD'00)*, volume 1954 of *LNCS*, pages 390–404. Springer-Verlag, 2000.

[5] S. Blom and S. Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *Int J Softw Tools Technol Transfer*, 2004.

[6] B. Bollig, M. Leucker, and M. Weber. Parallel model checking for the alternation free μ-calculus. In *Proceedings of the 7th International Conference on Tools and Algorithms for the*

Construction and Analysis of Systems (TACAS'01), volume 2031 of *LNCS*, pages 543–558. Springer-Verlag, 2001.

[7] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In *Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science (FST TCS'01)*, volume 2245 of *LNCS*, pages 96–107. Springer-Verlag, 2001.

[8] I. Černá and R. Pelánek. Distributed explicit fair cycle detection (set based approach). In *Proceedings of the 10th International SPIN Workshop on Model Checking of Software (SPIN'03)*, volume 2648 of *LNCS*, pages 49 – 73. Springer-Verlag, 2003.

[9] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[10] H. Garavel, R. Mateescu, and I.M Smarandache. Parallel State Space Construction for Model-Checking. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *LNCS*, pages 200–216. Springer-Verlag, 2001.

[11] P. Godefroid and P. Wolper. A partial approach to model checking. *Logic in Computer Science*, pages 406 – 415, 1991.

[12] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, May 1994.

[13] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 20–35. Springer-Verlag, 2000.

[14] T. Heyman, O. Grumberg, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 54–66. Springer-Verlag, 2003.

[15] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proceedings of the 6th International SPIN Workshop on Model Checking of Software (SPIN'99)*, volume 1680 of *LNCS*, pages 22–39. Springer-Verlag, 1999.

[16] R. Nalumasu and G. Gopalakrishnan. PV: A model-checker for verifying LTL-X properties. In *Fourth NASA Langley Formal Methods Workshop*, pages 153 – 161. NASA Conference Publication 3356, 1997.

[17] R. Palmer and G. Gopalakrishnan. A distributed partial order reduction algorithm. In *Proceedings of Formal Techniques for Networked and Distributed Systems - FORTE 2002, 22nd IFIP WG 6.1 International Conference Houston*, volume 2529 of *LNCS*, page 370. Springer-Verlag, 2002.

[18] R. Palmer and G. Gopalakrishnan. Partial order reduction assisted parallel model-checking (full version). Technical report, University of Utah, August, 2002.

[19] D. Peled. All from one, one from all: on model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93)*, volume 697 of *LNCS*, pages 409–423. Springer-Verlag, 1993.

[20] D. Peled. Combining partial order reductions with on-the-fly model checking. *Formal Methods in System Design*, 8(1):39 – 64, 1996.

[21] D. Peled. Ten years of partial order reduction. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 17–28. Springer-Verlag, 1998.

[22] U. Stern and D. L. Dill. Parallelizing the Murφ verifier. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 256–267. Springer-Verlag, 1997.