Partial Order Reduction Assisted Parallel Model-Checking

Robert Palmer and Ganesh Gopalakrishnan*
{rpalmer,ganesh}@cs.utah.edu,
http://www.cs.utah.edu/formal_verification/

University of Utah, School of Computing

Abstract. Partial order reduction helps improve the performance of a (sequential) model-checker by eliminating the interleaving of independent actions. In this paper, we show how to combine partial order reduction and parallel distributed model-checking. We point out that an appropriate partial order reduction algorithm is to be chosen to avoid sequentializing an otherwise parallelizable activity. We propose such an algorithm and show that our algorithm can reduce the number of states generated and limit unnecessary communication between network nodes. We discuss the implementation of these ideas as well as preliminary experimental results.

1 Introduction

Parallel processing can help speed-up model-checking by providing many CPUs that can work on the problem, providing higher aggregate memory capacity, as well as memory bandwidth. While the problem remains exponential, the overall performance can go up by significant factors. In this paper, we focus on enumerative model-checkers which are widely used to verify many classes of models, such as high-level cache coherence protocols and Java software models, in which models containing thousands of variables and involving global dependencies – such as processor IDs passed around in messages – are to be verified. Such models have never been shown to be capable of being efficiently represented or manipulated using BDDs. There are many previous attempts to parallelize enumerative model-checkers such as SPIN [1] and Murφ [2]; see our Related Work section for a brief survey.

Among the scores of methods used to reduce state-explosion, a prominent method - partial order reduction - exploits the fact that whenever a set of independent actions $\{o_1, \ldots, o_n\}$ arise during exploration, they need not be explored in more than one linear order. Given the widely recognized importance of partial order reduction, efficient methods to realize it in a parallel context must be explored. This topic has, hitherto, not been studied extensively. This paper is our first attempt to ameliorate the situation. We first motivate why partial order reduction and parallelism are two concepts that are closely related. We then briefly recap our partial order reduction algorithm "Twophase" implemented in our SPIN-like model-checker "PV," and summarize why Twophase is often more efficient than partial order reduction algorithms that use in-stack checking as the means of realizing the proviso condition (see [3] as well as our website for a full discussion). In this paper, we show that Twophase extends naturally to a parallel context - more readily than algorithms that use in-stack checking - and also leads to a very natural task-partitioning method.

1.1 Partial order reduction and parallel processing

The earliest identifiable connection between parallel processing and the central idea behind partial order reduction (long before that term was coined) appears in Lipton's work of 1975 on optimizing P/V programs [4] for the purpose of efficient reasoning. In that work, Lipton identifies *left* and *right* movers – actions that can be postponed without affecting the correctness of reasoning. Additionally, in the parallel compilation

* Supported by NSF Grants CCR-9987516 and CCR-0081406, and a gift from the Intel Corporation. The authors wish to thank Ratan Nalumasu who wrote the first version of PV, Hemanthkumar Sivaraj for his help with MPI, and Omid Saadati for developing the PV to Bandera link.

literature (e.g., [5] for an example), it has been observed that by identifying computations that "commute," one can schedule these computations for parallel evaluation without interference.

Despite the above connections, if a partial order reduction algorithm involves a sequentializing step, it can significantly reduce available parallelism. Such a sequentializing step is present in current partial order reduction algorithms such as the one used in SPIN [6, 7]. This is the *in-stack*[1] check used as a sufficient condition to eliminate the *ignoring* problem (a problem where some process P_j may be enabled everywhere in a cyclic execution path but never moved along the path, thus causing missed, "ignored," states). Using the 'in-stack' check, if the next states generated by moving a process P_i result in a state in the global DFS stack, an *ample set* (a commuting set of actions) cannot be formed out of the actions of P_i. The in-stack check is involved in every step of SPIN's partial order reduction algorithm. In a parallel setting, this could translate into considerable communication to locate the processes over which the stack is spread, perform the in-stack check, and resume the search. For this reason, in past work, a "worst case" assumption is used to gain some limited reductions. The assumption is that any successor state held outside the node is assumed to be currently in the search stack. This insures that the ignoring problem is dealt with, but may cause significant loss in reduction [8].

1.2 Background on Twophase

We created Twophase after realizing that SPIN's algorithm for partial order reduction can, due to the in-stack check based proviso, miss out on reduction opportunities [3]. There are also added complications when realizing nested DFS based LTL-x checking [9], as explained in [10] - essentially requiring a 'proviso-bit' to convey information from the outer DFS to the inner DFS.

In contrast to SPIN's algorithm, Twophase is a *much* simpler algorithm, is easier to prove correct, obtains superior performance on a large class of examples, and does not involve any sequentializing steps such as the in-stack check. Twophase works as follows (see Figure 1). Assume that search starts at the initial state s_0. When the search is at state s_i, Twophase checks to see whether some process P_i has a singleton *ample set*[2]. If this is *not* the case, Twophase performs *full expansion* of state s_i and recursively invokes itself on all the successors (this entire step is called Phase-2). Suppose, however, that there is a *singleton* ample set with respect to process P_i - essentially, P_i has a commuting transition t_i. Then Twophase enters its Phase-1, generates state $s_j = t_i(s_i)$, checks for invariants, and if satisfied, continues as follows. If state s_j has a singleton ample set with respect to process P_j – meaning that P_j has a commuting transition t_j (note that $P_i = P_j$ is possible) – the execution remains in Phase-1, and Twophase continues at state $t_j(s_j)$ as it did at state s_i. While in Phase-1, PV keeps adding the visited states into a visited states set $list$, and checks for re-visitations into $list$. When such a re-visitation is found, Twophase continues with the next process (to see if it has a singleton ample set, as indicated by the `goto` to `NEXT_PROC` in Figure 1). *This avoids the ignoring problem without resorting to an in-stack check.* Finally, when there is no process with a singleton ample set at state s, Twophase returns from the call to `phase1`, adds $list$ to the hash table, and invokes itself recursively for each successor of s, as explained under `''Phase 2: Classic DFS''`.

The algorithm of Figure 1 can be suitably modified for nested DFS based LTL-x model-checking. In doing so, it does not require the proviso-bit to convey information from the outer DFS to the inner DFS. While this issue is not germane to this paper where we pursue only safety model-checking, a later extension of Twophase for LTL-x in a parallel setting would reap the benefits of its simplicity.

While in Phase-1, Twophase offers *three* distinct selective state-caching options that decide which of the states are added to $list$:

- *SaveAll:* This option says *turn off state-caching.* All states generated in Phase-1 are added to $list$.
- *SaveBackEdge:* This option says *enter a state $s_l = t_k(s_k)$ into $list$ only if in going from s_k to s_l, a ranking function* – for example, the value of the control state – *decreases.* This option obtains nearly the full benefit of selective state caching, and does not suffer from *ignoring*.

[1] "The" in-stack check is a misnomer - this check is done differently for safety-preserving and liveness-preserving reductions. To simplify things, we ignore such variations.

[2] Note that the singleton ampleset condition is exactly the one needed for CTL*-x preserving reductions.

```
model_check() {
  V_r := φ;  /* Hash table */
  Twophase(initial_state);
}

phase1(in) {
  local olds, s, list;
  s := in;
  list := {s};
  foreach process P do
    while (singleton_ampleset(s, P))
      /* Let t be the only enabled
       * transition in P at s */
      olds := s;
      s := t(olds);
      if (s ∈ list)
        goto NEXT_PROC;
      end if
      list :=  list + {s}; [1]
    end while;
    NEXT_PROC: skip
  end foreach;
  return(list, s);
}

Twophase(s) {
  local list;
  /* Phase 1 */
  (list, s) := phase1(s);
  /* Phase 2: Classic DFS */
  if s∉V_r then
    V_r := V_r + all states in list + {s};
    foreach enabled transition t do
      if t(s) ∉ V_r then
        Twophase(t(s));
      end if;
    end foreach;
  else
    V_r := V_r + all states in list;
  end if;
}
```

Fig. 1. The Twophase algorithm

Example	SPIN	Save All	Save Back Edge	Save None
Leader Election	1016380	1134651	684112	477570
PipeInt	1803530	1363019	1363019	1363019

Fig. 2. Statistics of PV and SPIN on two example models

- *SaveNone:* This option is an extreme form of state caching where we enter *none of the Phase-1 states into set* $list$. This option makes PV a semi-algorithm, as we can loop on a state generated within Phase-1. In practice, however, PV actually finishes, reducing the number of states slightly over *SaveBackEdge*s. Verification is sound whenever *SaveNone* finishes.

Figure 2 summarizes some of our results of two example models. The first is the leader election protocol as given in [6]. The PipeInt model is a Java based model generated by the Bandera[11] tool to which PV has been attached as a back-end. The table shows the number of states generated by PV using the various options described in this paper on these two models[3]. Further experimentation and comparison with the SPIN model checker in a sequential setting is available in [12] and from our website.

1.3 Twophase in a Parallel Context

In a parallel context, we choose to realize safety model-checking. Our motivations for this decision are: (i) liveness checking is inherently sequential; (ii) liveness can, in many practical situations, be checked through bounded model-checking; (iii) this will be part of our future work. In a safety model-checking setting, Twophase offers a very natural method for parallelizing activities:

[3] While the the number of states generated by SPIN is lower for these examples compared to those obtained under "SaveAll" for PV, for many of our examples, PV generated fewer states even with this option.

- While in `phase1`, we can avoid communication between the parallel nodes altogether. In contrast, in previous works that employ a uniform hashing function for work distribution (e.g., [13]), on the average $(N-1)$ out of the N next-states are sent elsewhere, thus causing a flurry of communication per model-checking step. Sequences of states that are generated while in `phase1` are akin to a *dynamically determined* `dstep` – essentially, a sequential chunk of computation that need not be interleaved with other computations. In a parallel setting, these are also the "quiet" epochs where communication need not occur.
- At the end of `phase1`, when we do the "Classic DFS," we can perform state distribution, and resume with a `phase1` step for each of the distributed states.

To sum up, the main observations of this paper are threefold. First, the absence of the in-stack check in Twophase makes its extension to a distributed setting highly parallel. We need no "worst case" assumptions as in [8]. Second, it also leads to reduced communication activities during `phase1`, and also supports selective state caching during `phase1`. Last but not least, if we were to extend Twophase for distributed LTL-x checking, it would also not need the proviso bit and the associated complications.

1.4 Roadmap

In the next section, we briefly survey related work. In the remainder of the paper, we present the algorithm for parallel model checking using the Twophase partial order reduction algorithm. We present a method for further reducing the memory requirement of a distributed model checking computation. We will discuss some of the implementation details and give preliminary results on the Leader Election and PipeInt Promela models.

1.5 Related work

Work in parallel and distributed model checking can be divided into the categories of *explicit state representation based* and *symbolic state representation based*.

Explicit state Safety model checking: Most work on distributed model checking focus on safety model checking. In [14], Stern and Dill report their study of parallelizing the Murϕ Verifier [2]. It originally ran on the Berkeley Network of Workstations (NOW) [15] using the Berkeley Active Messages library. It was subsequently ported to run on the IBM SP2 processor. Murϕ is a safety-only explicit state enumeration model checker. In its parallel incarnation, whenever a state on the breadth-first search queue is expanded, a uniform hashing function is applied to each successor s to determine its "owner" – the node that records the fact that s has been visited, and pursues the expansion of s.

We [16] have recently ported parallel Murϕ from Active Messages to the popular MPI [17] library. Despite our relative inattention to performance for reasons of expediency, our speed-up figures for runs on the Testbed are very encouraging [18]. The largest model we ran far exceeds the sizes run by Stern and Dill.

In [19], a distributed implementation of the SPIN [20] model checker, restricted to perform safety model checking, and similar to [14], is described. Their first innovation is in state distribution. They exploit the structure of SPIN's state representation and reduce (heuristically) the number of times a state is sent to other nodes. In addition, they employ look-ahead computation to avoid cases where a state is sent elsewhere, but very soon generates a successor that comes back to the original node. Their algorithm is also compatible with partial order reduction, although the reported results to date do not include the effects of this optimization. Their examples are standard ones such as 'Bakery' and 'Philosophers' running on upto four nodes on 300MHz machines with 64M memory. In [21], the algorithm of [14] is adapted to Uppal, a timed automaton model-checker, and applied to many realistic industrial-scale protocols, running on 24, 333MHz Sun Enterprise machines. Several scheduling policies are studied along with speed-up results.

In [22], parallel state space construction for labeled transition systems (LTSs) obtained from languages such as LOTOS is described. They use a cluster of 450MHz machines of upto 10 processors, each with 0.5GB

of memory. They use the widely supported `Socket` library. They obtain speedups on most examples (industrial bus protocols) and perform analysis of the effects of communication buffers on overall performance.

In [23], issues relating to software model checking and state representation are discussed. A large number of load distribution policies are discussed, and preliminary experimental results are reported. Many of these ideas are adaptations of techniques from their original work [19] to work well in the context of a software model such as Java.

LTL-x model checking: Several works go beyond state space reachability and attempt the distributed model checking of more expressive logics. In [24], the authors build on [19] and create a distributed LTL-x model checker. The main drawback of their work is that the standard [9, 25] nested depth-first search algorithm employed to detect accepting (violating) Büchi automaton cycles tends to run sequentially in a distributed context, as the postorder enumeration of the "seed" states is still essential. They ameliorate the situation slightly by employing a data-structure called DepS that records how states were transported from processor to processor, and gathering the postorder numbering of the seed states in a distributed manner. However, the seed states still end up in a central queue, and are processed sequentially. A small degree of pipelining parallelism appears possible between the inner depth-first search on the "left half" of the search tree and the outer depth-first search on the "right half" of the search tree. Their paper reports feasibility (without actual examples) on a nine 366MHz Pentium cluster.

In [26], Büchi acceptance is reduced to detecting negative cycles (those that have a negative sum of edge weights) in a weighted directed graph. This reduction is achieved by attaching an edge-weight of '-1' to all outgoing edges out of an accepting state, and a weight of '0' to all other edges.

Symbolic state In [27], scalability in parallel reachability analysis is studied. Their system carries out BDD based reachability analysis on a distributed platform. Their primary objective is to obtain the benefits of the large combined amounts of memory in a distributed context. Their study includes various slicing heuristics as well as their performance on upto 32, 266MHz RS6000 machines, each with 256MB memory, connected by a 16Mb/s token ring. Their distributed implementation could reasonably well utilize the available memory (the memory overhead being close to a factor of three), and in one case reached 35 steps in the fixed-point iteration, compared to 18 steps on a uniprocessor with 768MB memory. In [28], a distributed on-the-fly symbolic model checking algorithm for RCTL, a simplified temporal logic used for hardware verification and test generation, is presented. The machines used are identical (at least in features) to that used in [27]. Speed-up characteristics for various circuit and hardware benchmarks are presented. Finally, in [29], a distributed symbolic model checking algorithm for μ-calculus, its correctness proof, as well as sources of scalability are presented.

The use of parallel and distributed machines for hardware verification in contexts other than model checking have been widely researched. Of recent heightened interest are Boolean Satisfiability methods whose parallelization is also under study [30].

2 Parallel Twophase Algorithm

Sequential SPIN and PV both use a depth first search for model checking safety properties. The distributed version of SPIN in [8] maintains the depth first search strategy. Since Twophase is not dependent upon a search stack, the distributed version of PV can be implemented using a breadth first search strategy, which is what we do. This is a real advantage since BFS is inherently parallel while DFS is inherently sequential (P-complete).

The state space is partitioned statically using a hashing function. This function attempts to uniformly divide the state space among the participating network nodes. When a global state s is reached in the search, Phase-2 is entered, and the successors of s are computed using the partitioning function and sent to the owned nodes, or (if owned locally) enqueued locally. Each of these nodes are expanded using Phase-1. Figure 3 shows the complete algorithm. Phase-1 remains the same as the sequential depth first version. Note that the ignoring problem is avoided using a completely local list.

```
10 procedure DistributedTwophase(myid)
11   s = initial state
12   V = { } /* Set of visited states */
13   Q = { } /* Queue of states waiting to be expanded */
14   list = { } /* The ‘‘mini-list’’ */
15   s = Phase1(&list, s)
16   V = V + states in list
17   i = partition(s)
18   if i == myid
19     Q = Q + {s}
20   else
21     send(s, i)
22   while search is not complete
23     s = Q head
24     t = set of enabled transitions in s
25     for each transition in t
26       if t(s) not in V
27         s = Phase1(&list, t(s))
28         V = V + states in list
29         i = partition(s)
30         if i == myid
31           Q = Q + {s}
32         else
33           send(s, i)
34         end if
35       end if
36     end for each
37   end while
38 end procedure
```

Fig. 3. The Distributed BFS version of Twophase.

Partial order reduction can be easily disabled in the distributed version of the Twophase algorithm as well. By not executing line 27 of Figure 3 a classical distributed breadth first search remains.

In many cases, Phase-1 executes transitions that will cross the boundary of the state space partition. Thus at least one, and as many as all of the states entered into the list are not “owned” by the node performing the computation. In this case it is not necessary for the local node to retain these states in the state store. They are then marked for removal at the termination of Phase-1. This technique can be used with all of the selective state caching variants described above. This is termed the “drop non-local states” or simply “Drop States” optimization.

With the elimination of a search stack, a technique for generating error trails has also been implemented. When a state (s') is entered into the state store, a memory reference to the state store location containing the global predecessor (s) of the new state (s') is also entered, along with a small constant amount of information about the state (s). This works quite naturally in a distributed setting since the only additional information needed is the rank of the node that created the state (s).

When an error state is visited the state store can then be traversed from the error state, across global states, to the initial state. If it is necessary to traverse multiple partitions of the state space, the error trail generated so far is packed and transmitted to the next process in the trail. An error trail can then be simulated in a sequential setting.

Finally termination detection of an exhaustive search (where no error is found), is done using the Dijkstra-Scholten algorithm for stable condition detection in a diffused computation[31].

3 Experimental Results

Results of model checking the Leader Election protocol, as shown in [6], along with the Java based, Bandera generated, PipeInt model are shown from this section onwards. All experiments done under "No Twophase" are with partial order reduction turned off. In most cases, such runs do not finish. We only report the number of states generated (runtime, message count, etc, are reported for a few examples now - we will have a full account in our final version). In all the experiments, the *number of processes* are the number of process in the model – all runs were using one Unix process per node.

Figure 4 shows the number of states placed in the state store while performing an exhaustive search both with and without partial order reduction as described above. We also include the statistics on the heuristic used to determine which states to enter into the list. Notice that we can use the Save All and Save Backedges options in combination with Drop States or its opposite, namely Save States. In all these cases,

Number of Network Nodes	No Twophase	Save All	Save BackEdge	Save All	Save BackEdge
		SaveStates		DropStates	
6 Processes					
1	221239	47086	33166	47086	33166
2	221239	53791	36449	34398	25327
4	221239	58274	38397	31221	23723
8	221239	66777	40480	20644	14739
7 Processes					
1	1719197	243704	169637	243704	169637
2	1719197	277614	185567	185052	135139
4	1719197	314701	199870	150847	118028
8	1719197	334506	211079	135169	106307
8 Processes					
1	n/a	1243666	857554	1243666	857554
2	n/a	1416659	963177	909948	671314
4	n/a	1582202	1023171	747522	584468
8	n/a	1727274	1084586	617526	497252

Fig. 4. Number of states generated for the Leader Election Protocol

Number of Network Nodes	No Twophase	Save All	Save Back Edge	Save All	Save Back Edge
		Save States		Drop States	
1	nc	nc	349708	nc	349708
2	nc	nc	375506	nc	266106
4	nc	nc	375506	nc	266106
8	nc	4310733	377906	1298669	263186

Fig. 5. Number of states generated exploring the PipeInt Bandera model

The state count for the PipeInt model is shown in figure5. The model is much larger than the Leader Election protocol (the former has a 3000-bit state-vector and the latter a 500-bit state vector).

The number of messages between network nodes is shown in Figure 6 and 7. By communicating only global states, a natural consequence of using partial order reduction is a reduction of communication between network nodes.

Number of Network Nodes	Not Reduced	Reduced
6 Processes		
2	214063	11343
4	277764	14795
8	418088	25217
7 Processes		
2	1855347	63727
4	2440071	94509
8	3269492	109306
8 Processes		
2	nc	358795
4	nc	492444
8	nc	603660

Fig. 6. Number of messages passed for Leader Election Protocol

The Leader Election protocol with 8 processes in the model does not finish on eight machines without partial order reduction. Similarly, the PipeInt model requires much more memory to exhaustively search the state space without partial order reduction.

Number of Network Nodes	Not Reduced	Reduced
2	nc	96326
4	nc	128646
8	nc	167988

Fig. 7. Number of messages passed exploring the PipeInt Bandera model

The experiments were performed on a computational cluster of eight FreeBSD workstations. Each has 512MB memory and one 850MHz Intel PentiumIII CPU. The MPICH[32] implementation of the MPI standard is used for message passing between network nodes. Figure entries showing “nc” indicate the computation was Not Complete. In each of these cases there was not enough memory to successfully generate the entire state graph. (We have not resorted to any hash-compaction techniques yet.)

4 Conclusions

We have presented a distributed partial order reduction based safety verification algorithm that is a variant of the sequential Twophase [3] algorithm. The advantages of this algorithm are:

- It avoids the inherently sequential in-stack check.
- Since no stack is needed, we can use BFS (which is inherently parallel) as opposed to DFS.
- It allows natural task partitioning that also reduces the amount of communication.
- It supports selective state caching in conjunction with a “drop non-local states” optimization.

- The advantages are obtained in protocol verification and distributed software model-checking in conjunction with Bandera.

Our future work will include application-level checkpointing to be able to suspend and/or rerun crashed parallel model-checks, hash compaction, and possibly symmetry reduction also.

References

1. Gerard Holzmann. *Design and Validation of Computer Protocals*. Prentice Hall, 1991.
2. David Dill. The murϕ verification system. In *Computer Aided Verification (CAV)*, pages 390–3, 1996.
3. Ratan Nalumasu and Ganesh Gopalakrishnan. An efficent partial order reduction algorithm with an alternative proviso implementation. *Formal Methods in System Design*, 20(3):231–247, May 2002.
4. Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12), December 1975.
5. Martin Rinard and Pedro Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):1–47, November 1997.
6. Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000.
7. G.J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. 12th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, Orlando, Fl., June 1992.
8. Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with SPIN. In *Proc. of the 5th International SPIN Workshop*, volume 1680 of *LNCS*. Springer-Verlag, 1999.
9. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. pages 233–242, June 1990.
10. G. J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, 1994. Chapman & Hall.
11. James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
12. Ganesh Gopalakrishnan, Ratan Nalumasu, Robert Palmer, Prosenjit Chaterjee, and Ben Prather. Performance studies of pv: an on-the-fly model-checker for ltl-x featuring selective state caching and partial order reduction. Technical Report UUCS-01-004, The University of Utah, School of Computing, January 2001.
13. Ulrich Stern and David L. Dill. Parallelizing the Murϕ verifier. In *Computer Aided Verification*, pages 256–278, 1997.
14. Ulrich Stern and David Dill. Parallelizing the Murϕ verifier. *Formal Methods in System Design*, 18(2):117–129, 2001. (Journal version of their CAV 1997 paper).
15. Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for networks of workstations: Now. *IEEE Micro*, February 1995.
16. Hemanthkumar Sivaraj. MPI port conducted in October 2001. Personal Communication.
17. Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1996. ISBN 1-55860-339-5.
18. Prosenjit Chatterjee, Hemanthkumar Sivaraj, and Ganesh Gopalakrishnan. Shared memory consistency protocol verification against weak memory models: refinement via model-checking. To appear in CAV'02.
19. Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with spin. In *Proceedings of the 5th International SPIN Workshop*, pages 22–39, 1999.
20. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
21. G. Behrmann, T.S. Hune, and F.W. Vaandrager. Distributed timed model checking - how the search order matters. In *Computer Aided Verification (CAV)*, pages 216–231, 2000. LNCS 1855.

22. Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel state space construction for model-checking. In *Proceedings of the 7th International SPIN Workshop*, pages 217–234, 2001. LNCS 2057.

23. Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *Proceedings of the 7th International SPIN Workshop*, pages 80–102, 2001. LNCS 2057.

24. Jiri Barnat, Lubos Brim, and Jitka Stribrna. Distributed ltl model-checking in spin. In *Proceedings of the 7th International SPIN Workshop*, pages 200–216, 2001. LNCS 2057.

25. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. Kluwer Academic Press, November 1999. (to be published).

26. Lubos Brim, Ivana Cerna, Pavel Krcal, and Radek Pelanek. Distributed ltl model checking based on negative cycle detection. In *Proceedings of the FSTTCS Conference*, 2001. Bangalore, India, December 2001. To appear.

27. Tamir Heyman, Danny Geist, Orna Grumberg, and Assaf Shuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Computer Aided Verification (CAV)*, pages 20–35, 2000. LNCS 1855.

28. Shoham Ben-David, Tamir Heyman, Orna Grumberg, and Assaf Shuster. Scalable distributed on-the-fly symbolic model checking. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 390–404, 2000. LNCS 1954.

29. Orna Grumberg, Tamir Heyman, and Assaf Shuster. Distributed symbolic model checking for μ-calculus. In *Computer Aided Verification (CAV)*, pages 350–362, 2001. LNCS 2102.

30. C. Madigan Y. Zhao, M. Moskewicz and S. Malik. Accelerating boolean satisfiability through application specific processing. In *Proceedings of the International Symposium on System Synthesis (ISSS)*. IEEE, October 2001.

31. E. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

32. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.