Simplified Distributed LTL Model Checking by Localizing Cycles

Alberto Lluch Lafuente
March 5, 2002

Institut für Informatik, Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee Geb. 051, D-79110 Freiburg, Germany
email:lafuente@informatik.uni-freiburg.de

Abstract. Distributed Model Checking avoids the state explosion problem by using the computational resources of parallel environments LTL model checking mainly entails detecting accepting cycles in a state transition graph. The nested depth-first search algorithm used for this purpose is difficult to parallelize since it is based on the depth-first search traversal order which is inherently sequential. Proposed solutions make use of data structures and synchronization mechanisms in order to preserve the depth-first order. We propose a simple distributed algorithm that assumes cycles to be localized by the partition function. Cycles can then be checked without requiring particular synchronization mechanisms. Methods for constructing such kind of partition functions are also proposed. The algorithm has a limited application since it highly depends on the monotonic behavior of the model to be checked.

1 Introduction

Automated verification methods like model checking [4] suffer from the state explosion problem. Computational resources, especially space, become the main limiting factors. Distributed model checking tackles with this problem by exploiting the amount of resources provided by parallel environments. For instance, a network of workstation can be used by distributing the state space among the network nodes so that larger state spaces can be explored. A speed-up is also be expected if the verification is done in parallel.

While early approaches to parallel and distributed model checking are limited to the verification of safety properties [3,11,13,9,2], recent works propose methods for checking liveness properties expressed in linear temporal logic (LTL) [10]. LTL Model checking mainly entails finding accepting cycles in a state transition graph which is done with the nested depth-first search algorithm. The correctness of this algorithm depends on the depth-first traversal of the state space. Since depth-first search is inherently sequential [12], additional data structures and synchronization mechanisms must be added to the algorithm. This requirements can waste the resources offered by the distributed environment. Moreover, formally proving the correctness of the resulting algorithms is not easy.

Our approach tries to keep the search algorithm as simple as possible. As in [11,10] the state space is distributed among the network nodes which perform the verification process. A partition function divides the state space into equivalence classes of states such that cycles exist only in within equivalence classes. A static analysis of the model and the specification is used to define a partition that localizes cycles into equivalence classes. Cycle detection is done within equivalence classes only, without requiring extra data structure or synchronization. Since the traversal order within equivalence classes of states is depth-first, the algorithm remains correct.

This paper is structured as follows. Section 2 gives the necessary background on automata based LTL model checking. Section 3 presents the distributed LTL model checking algorithm. Section 4 describes the methods that can be used in order to define partition functions that localize cycles. Related work is described in Section 5. Finally, the last section concludes the paper.

2 Automata-Based LTL Model Checking

Model checking is an automated formal verification method. A model of a system is analyzed for checking if it satisfies its specification or not. Specifications are typically given in temporal logics like linear-time temporal logic (LTL). Some LTL model checkers like SPIN [1] take an automata-based approach to model checking. Both the model and the specification are modeled with automata.

Büchi automata are often used since they allow to represent infinite behaviors. A Büchi automaton is a five tuple $\langle \Sigma, Q, \delta, Q_0, F \rangle$, where Σ is a finite alphabet, Q is the finite set of states, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $Q_0 \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of accepting states. Büchi automata run over infinite words. A run of a Büchi automata is accepting if and only if an accepting state appears infinitely often in the run. The *language* $\mathcal{L}(B)$ *accepted by the Büchi automaton* B is then the set of infinite words, over which all runs of B are accepting.

Checking that a model $\mathcal{M}$ satisfies its specification $\mathcal{S}$ consists on verifying if the language accepted by the model is included in that of the specification. Checking language inclusion $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{S})$ is equivalent to checking if the intersection $\mathcal{L}(\mathcal{M}) \cap \overline{\mathcal{L}(\mathcal{S})}$ is empty. In practice, it is more efficient to check emptiness of language intersection.

Checking emptiness of the intersection of two Büchi automata entails detecting accepting cycles in the state space of the synchronous product of both automata. Let M be a finite state automaton representing the model, and let N be the *never claim* automaton. The never claim is a Büchi automata that accepts the same language as the negation of the LTL formulae representing the specification, i.e. $\mathcal{L}(\mathcal{N}) = \overline{\mathcal{L}(\mathcal{S})}$. Automaton M is interpreted as a Büchi automaton in which *all* states are accepting.

The *synchronous product* $M \otimes N$ of M and N is defined as: $M \otimes N = \langle \Sigma, Q, \delta, Q_0, F \rangle$, where $Q = Q_M \times Q_N$, $Q_0 = Q_0^M \times Q_0^N$, $F = F_M \times F_N =$

[1] `netlib.bell-labs.com/netlib/spin`

$Q_M \times F_N$, and $((s_M, s_N), a, (s'_M, s'_N)) \in \delta'$ if and only if $(s_M, a, s'_M) \in \delta_M$ and $(s_N, a, s'_N) \in \delta_N$.

Büchi automata can be represented as directed graphs: the set of vertices is Q and the edges are labeled by the transition relation δ. Runs of the automaton over an infinite word correspond to infinite paths in the graph, and accepting runs to infinite paths containing infinite accepting cycles. An accepting cycle is defined as a cycle in which at least one state is accepting.

A strongly connected component (SCC) of a directed graph is a maximal set of vertices, such that each vertex in the set is reachable from each other vertex of the set. It is not difficult to show that pairwise reachability is an equivalence relation such that the set of nodes of a graph can be partitioned into equivalence classes of strongly connected components. An important consequence of the definition of SCCs is that all vertices of a cycle belong to the same SCC. In the following we write $scc(s)$ to denote the SCC to which a state s belongs.

procedure $dfs1(s)$
    $V \leftarrow V \cup \{s\}$;
    for all successors s' of s **do**
      if $s' \notin V$ **then** $dfs1(s')$;
    if $accept(s)$ **then** $dfs2(s)$;

procedure $dfs2(s)$
    $flag(s)$;
    for all successors s' of s **do**
      if s' on $dfs1 - stack$ **then terminate**;
      else if s' not $flagged(s)$ $\underline{\text{and } \pi(s) = \pi(s')}$ **then** $dfs2(s')$;

Fig. 1. Nested depth-first search algorithm.

Checking emptiness of a Büchi automaton consists on finding accepting cycles in its state space. This is done by the nested depth-first search algorithm depicted in Figure 1. The underlined part of the last line is not part of the original algorithm. It will be used in a next section to prove correctness of the distributed algorithm and has to be ignored at this point. Two depth-first search procedures are used each with its own stack. The first search procedure ($dfs1$) explores the state space in a depth-first manner. Visited states are stored in set V. When the first search backtracks from the exploration of an accepting state s the second search procedure ($dfs2$) is invoked. This search explores the successors of s that have not yet been checked for cycles. States visited by the second search are marked with a special flag. If during the second search a state is found on the stack of the first search, an accepting cycle has been found and the algorithm terminates.

3 Distributed LTL Model Checking

This section presents the distributed version of the nested depth-first search algorithm. We take the approach of [11] as basis. The state space to be explored is distributed between the network nodes. Every node owns a subset of the states. Each node uses a work queue to store the states that the node has still to explore. Nodes extract states from this queue and compute their successors. Successors states belonging to other nodes are sent to the corresponding queues, while successors belonging to the same node are explored as in the sequential algorithm. When all nodes are idle and all work queues are empty the algorithm ends.

procedure $start()$
  $V(my\_pid) \leftarrow \emptyset$;
  $U(my\_pid) \leftarrow \emptyset$;
  if $\pi(start\_state) = my\_pid$ **then**
   $U(my\_pid) \leftarrow U(my\_pid) + start\_state$;
  $visit()$;

procedure $visit()$
  do
   wait until not $empty(U(my\_pid))$;
   $s \leftarrow extract(U(my\_pid))$;
   $dfs1(s)$;

procedure $dfs1(s)$
  $V(my\_pid) \leftarrow V(my\_pid) \cup \{s\}$;
  for all successors s' of s **do**
   if $\pi(s') = my\_pid$ **then**
    if $s' \notin V(my\_pid)$ **then** $dfs1(s')$;
   else $U(\pi(s')) \leftarrow U(\pi(s')) + s'$;
  if $accept(s)$ **then** $dfs2(s)$;

procedure $dfs2(s)$
  $flag(s)$;
  for all successors s' of s **do**
   if s' on $dfs1 - stack$ **then terminate**;
   else if s' not flagged and $\pi(s) = \pi(s')$ **then** $dfs2(s')$;

Fig. 2. Distributed nested depth-first search.

3.1 Algorithm

Figure 2 illustrates the pseudo-code for the distributed nested depth-first search algorithm. The algorithm assumes that cycles are localized by the partition function. Each node executes the same code, where the node identifier is denoted by my_id. Procedure $start$ is first executed. It initializes the set V used to store the visited states and the work queue U. Work queues are set to empty except for the node owning the start state. Procedure $visit$ is then called. This procedure manages the work queue by extracting states and calling the first search procedure $dfs1$. The first search uses the partition function π to determine if a successor s must be recursively explored or sent to another node. As in the nested depth-first search algorithm, the second search $dfs2$ is invoked when the algorithm backtracks from the exploration of an accepting state s commonly called *seed*. At this point it cannot be assumed that all successors of s have been explored. Since all states of a strongly connected component belong to the same equivalence class it is however sufficient for the correctness of the algorithm to observe states reachable from s and belonging to the same strongly connected component have been explored. Correctness of this algorithm will be explained in a next section. The second search explores only states belonging to the same equivalence class of the seed, since no cycle entails two different equivalence classes. Termination detection, keeping track of of traces, and other issues can be done as in [11].

3.2 Correctness

The algorithm presented in the previous section is correct under the assumption that the partition localizes strongly connected components of the state space. We first show that without this assumption, the algorithm has two serious problems. First, if states belonging to he same SCC are explored in parallel, cycles might be missed. This can be illustrated with the example taken from [10] and depicted in Figure 3. If a nested search starts in state A and flags state C, a nested search starting from state B will not be able to detect the cycle B, C, D, B.

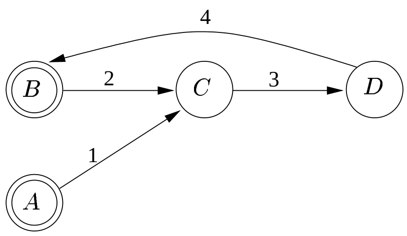

Fig. 3. Problems by parallel nested search.

The second problem arises when the states of a cycle are not explored in depth-first search traversal order. The example depicted in figure 4 and also taken from [10] shows this. Since the depth-first search order is not preserved, a nested search could start from state A before a doing it from state C, missing then the cycle C, D, B, C.

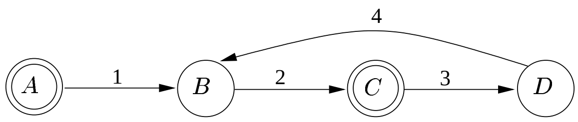

Fig. 4. Problems by non-depth-first order.

By assuming that all states within a cycle belong to the same equivalence classes both problems are avoided, since states belonging to the same equivalence classes are explored sequentially and in depth-first order.

We first show correctness of the nested depth-first search of Figure 1 including the underlined part. With this part the only difference with the original nested depth-first search algorithm is that the second search is restricted to explore states of the same equivalence class as the seed. We have to show that with this restriction the second search of the algorithm does not miss any accepting cycle. The proof is similar to that of the nested depth-first search described in [4].

Suppose the contrary. Let s be the first accepting state from which the second search starts but fails to find a cycle even though one exists. In the moment in which the second search starts from s there is at least one flagged state on a cycle through s. Let r be the first such state, and let s' be the state from which the second search that flagged r was started. In the algorithm the second search remains in the same equivalence class from which the search was started. Therefore s, r and s' must belong to the same equivalence class, i.e. $\pi(s) = \pi(r) = \pi(s')$. According to our assumptions, the second search from s' was started before the second search from s. There are two cases:

1) The state s' is reachable from s. Then there is a cycle $\langle s' \to \ldots \to r \to \ldots \to s \to \ldots \to s' \rangle$ that could not have been found previously, otherwise the algorithm would already have terminated. However, this contradicts our assumption from which the second search missed a cycle.

2) The state s' is not reachable from s. If s' appears on a cycle, then a cycle was missed before starting the second search at s. According to the assumption, s is reachable from r and, subsequently, s is reachable from s'. Thus, if s' does not occur on a cycle, we must have discovered and backtracked from s in the first search before backtracking from s'. Hence, according to the algorithm, we must have started a second search from s before starting it from s'. This contradicts the assumptions.

Correctness of the distributed algorithm follows from the fact that states within a SCC are explored in depth-first search order, so that the previous reasonings still hold.

4 Localizing Cycles

The algorithm presented in the previous section requires that the partition function distributes the state space such that all states of a strongly connected component (or equivalently a cycle) belong to the same equivalence class. We show how the structure of the model and the specification can be used in order to define such a function.

4.1 Exploiting the Specification

Recall that in LTL model checking specifications given as LTL formulae are translated into a negated Büchi automaton called never claim. We illustrate with an example how the structure of the never claim can be exploited.

The never claim automaton for a response property depicted in Figure 5 has two states n_0, n_a. Each state has a self transition. Each of these local cycles can *generate* cycles in the state space of the synchronous product. However, there is no transition from state n_a to state n_0. Global states can be divided in two subsets P_0 and P_a, such that P_0 is formed by those global states in which the never claim is in its local state n_0, while P_a contains the states where the never claim is in state n_a. There is no possible transition from P_a back to P_0 since there is no transition from n_a to n_0, and therefore global cycles can only exist either in P_0 or in P_a. In following lines we generalize this strategy.

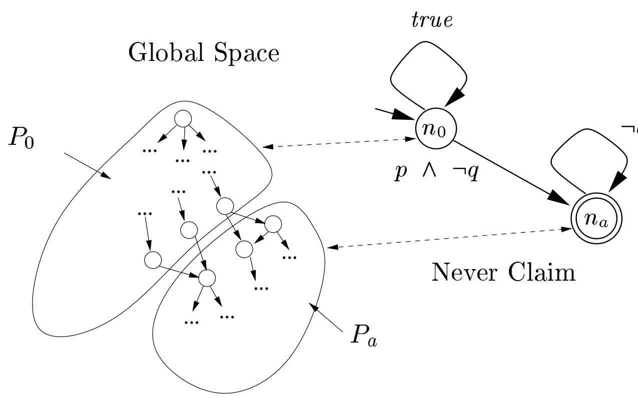

Fig. 5. Never claim automaton for a response property (right) and corresponding global state space of the synchronous product (left).

Let Q be the set of states of $M \otimes N$. We define a partition function π from Q onto $\{0, \ldots, k\}$ in such a manner that two states belong to the same equivalence class if and only if the state component of N belongs to the same SCC in the state transition graph of N. More precisely, if $s = (s_M, s_N) \in Q$ and $i = scc(s_N)$ then $\pi((s_M, s_N)) = i$. Obviously, π defines a partition of equivalence classes $P_0, \ldots P_k$ of Q, where $P_i = \{s \in Q \mid \pi(s) = i\}$, $i \in \{0, \ldots, k\}$.

It is easy to show that π distributes the state space in such a manner that cycles only exist within equivalence classes. If there is a cycle C in Q, then π partitions the states in Q in such a manner that all states of the cycle belong to the same equivalence class in Q.

Let C be a cycle in the global state transition graph. We have that $C = (s_0, s_1, \ldots, s_n)$ with $s_n = s_0$ and $(s_i, a, s_{i+1}) \in \delta$ for all $i \in \{0, \ldots, n-1\}$. Therefore, since $s_i = (s_i^M, s_i^N)$ and $s_i^N \in N$, $i \in \{0, \ldots, n\}$, a cycle $C_N = (s_0^N, s_1^N, .., s_n^N = s_0^N)$ exists with $(s_i^N, a, s_{i+1}^N) \in \delta_N$ for all $i \in \{0, \ldots, n\}$. Hence, for all $s_i = (s_i^M, s_i^N)$ and $s_j = (s_j^M, s_j^N)$ in C we have $scc(s_i^N) = scc(s_j^N)$. This implies $\pi(s_i) = \pi(s_j)$ for all $s_i, s_j \in C$. Therefore all states of C belong to the same equivalence class.

The main drawback of this approach is that LTL specifications are rather simple in practice. Table 1 depicts the number of SCCs in never claim automata corresponding to typical specification patterns identified in [5]. A dash is present if the property express safety or liveness behaviors that do not require cycle detection. Translation from LTL to Büchi automaton has been performed with SPIN. In the worst case, the never claim for an existence globally property contains only one SCC which restricts the number of equivalence classes to one. Even if the never claim has many SCCs, there is no way to know a priori how the state space is distributed, so that good load balances are difficult to be achieved.

Pattern	Globally	Before	After	Between	Until
Absence	-	-	-	-	2
Universality	-	-	-	2	2
Existence	1	2	3	3	2
Response	2	4	4	5	6
Precedence	2	2	2	-	3

Table 1. Number of SCCs in never claims for typical specification patterns.

Computing the strongly connected components of the state transition graph of the never claim can be done with Tarjan's algorithm which runs in time linear with respect to the size of the graph. In addition, never claims are small in practice, so that the time overhead of this pre-computation can be ignored.

4.2 Exploiting the Processes

The structure of the model can also be exploited as explained in the previous section. Software systems are formed of many interacting components, that can be typically modeled by communicating finite state machines combined following an interleaving model. The model of the system is therefore constructed as the asynchronous product of various communicating automata. As a consequence a cycle in the asynchronous product entails at least one cycle in every automata (which may be of length zero). The Cartesian product of the SCCs of the finite state machines can be used as partition function. However, in practice, the state transition graphs of the processes of the system has one unique SCC. An alternative is to exploit further exploit the monotonic behaviors of some systems.

Suppose that one of the variables of the model is initialized with a certain value. Suppose that variable can monotonically change its value. It is clear that in the global state space no cycle will be present in which that variable has its initial value and changes it. Therefore the state space can be divided according to the range of values of that variable, such that cycles are localized. A static analysis can determine which variables are changed monotonically. For example, if an integer variable is monotonically incremented.

Let V be the set of variables of the model, T the set of all transitions and C denote the set of strongly connected components of the state transition graph of the processes. We denote the effect of a transition $t \in T$ on a variable $v \in V$ with a function $f : T \times V \to \{0, +, -, \star\}$, where 0 denotes no effect, $+, -$ respectively denote incrementing and decrementing effect and $\star$ denotes that the precise effect is difficult to analyze. For the sake of simplicity, we have considered numerical variables. Boolean variables can be treated as numerical variables and similar ideas can be applied to communication queues, by analyzing if the size of a queue grows or decreases monotonically.

Let $F(c, v)$ denote the overall effect of a strongly connected component $c \in C$ on a variable v. The range of F is the same as that of f. The value of $F(c, v)$ is $+$ $(-)$ if there exist at least one transition $t \in c$ such that $f(t, v) = +$ $(f(t, v) = +)$ and there is no transition $t \in c$ such that $f(t, v) \in -, \star$. If every transition $t \in c$ has no effect on c then $F(c, v) = 0$, otherwise $F(c, v) = \star$.

The algorithm depicted in Figure 6 is used to compute the set $V_p \subseteq V$ of variables that can be used for constructing a partition function. The algorithm first computes the set of strongly connected components by invoking procedure *Compute_C* which pseudo code is not included for the sake of brevity. It basically applies Tarjan's algorithm to the state transition graph of each component process. In a next step F is computed. The algorithm then computes the set V_p. The main idea is that a variable v can be included in V_p if there is at least a strongly connected component which effect on v is incrementing (respectively decrementing) and no other strongly connected component has complex or decrementing (respectively incrementing) effect on v.

Tarjan's algorithm can be computed in linear time with respect to the size of the graph. Since we need apply it for the state transition graph of every component process we have that *Compute_C* is $O(|T|)$. Computing F is $O(|T| \times$

```
procedure Compute_V_p
    Compute_C;
    Compute_F;
    V_p ← ∅;
    for v ∈ V do
      F_v ← 0
      for c ∈ C do
        if F(c, v) = ⋆ or F(c, v) ≠ 0 ∧ F(c, v) ≠ F_v then break;
        else F_v ← F(c, v);
      if F_v ∉ {0, ⋆} then V_p ← V_p ∪ {v};

procedure Compute_F
    F(c, v) ← 0, ∀c ∈ C, v ∈ V;
    for c ∈ C, t ∈ C do
      for v ∈ V do
        if f(t, v) = ⋆ or F(c, v) ∉ {0, f(t, v)} then
          F(c, v) ← ⋆; break;
        F(c, v) ← f(t, v);
```

Fig. 6. Analysis of monotony in the model.

$|V|$). The rest of the algorithm is also $O(|T| \times |V|)$. Finally we have that the time complexity is $O(|T| \times |V|)$. We must however consider that the size $|T| \times |V|$ is in practice much more smaller than the size of the global state space so that the time requirements of this computation can be ignored. Space complexity can also be ignored since the structures used by this algorithm are not necessary in the verification process.

The algorithm can detect monotonic behaviors in models. A good example for this is the leader election algorithm in which each process owns two boolean variables that are monotonically changed. We can therefore define $2n$ partition functions each with a range of size 2. By combining them as described in a next section we can divide the state space in 2^{2n} parts. Unfortunately typical reactive systems show no monotony. These systems are commonly composed by processes which have cyclic behaviors such that the system can always return to the initial state. In other words, there is a unique large strongly connected component in the global state space of the model. A good example is the model of the dining philosophers problem which structure cannot be exploited to partition the state space.

4.3 Combining Partition Functions

Applying the described strategies can deliver a set of partition functions that localize cycles. Given a pair of such functions π_1, π_2 a new partition π can be constructed by combining them such that π also localizes cycles.

Let R_1, R_2 be the range of π_1 and π_2 respectively. Function π is defined over the Cartesian product $R_1 \times R_2$ such that $\pi(s) = (\pi_1(s), \pi_2(s))$.

It is easy to show that π localizes cycles. Suppose the contrary. Let C be a cycle with at least two states u, v such and $\pi(u) \neq \pi(v)$. Since we have that $\pi(u) = (\pi_1(u), \pi_2(u))$ and $\pi(v) = (\pi_1(v), \pi_2(v))$, then $\pi_1(u) \neq \pi_1(v)$ or $\pi_2(u) \neq \pi_2(v)$. Therefore be have that cycle C contains states that do not belong to the same equivalence class of π_1 or π_2 which cannot occur since we assumed that both partition function localize cycles within equivalence classes.

4.4 Mapping Partition Functions into Network Nodes

Up to this point we have assumed that the same function that partitions the state space approximating strongly connected components is used to distributes the states among the network nodes. In practice, the number of network nodes is different than the size of the range of the partition function such that it is necessary to define a function Π that maps R into the set of network nodes $\{0..P\}$. The resulting algorithm uses then the mapping function Π in procedure *start* and $dfs1$ instead of π. The simplest approach to define a mapping is to use a hash function, but this do not guarantee an equilibrate load distribution.

5 Related Work

Parallel cycle detection has been studied in many different contexts. For instance, the authors [1] propose a three phase parallel algorithm for the detection of cycles in planar directed graphs. Unfortunately state transition graphs in model checking are not planar in general and this approach cannot be taken. Other works [6,8] propose divide-and-conquer strategies which are not appropriate for on-the-fly state space generation.

Parallel model checking approaches have been proposed by some authors, for instance [3,11,13,9,2,10,7]. While most of them perform only reachability analysis and therefore restrict to the verification of safety properties, the solution of [10] applies to liveness LTL properties. Their approach uses data structures and synchronization mechanisms in order to guarantee the depth-first search traversal order necessary for the correctness of the nested depth-first algorithm. Space complexity is on average linear in the size of the state space and the factor given by non-determinism of the model. Contrary to us, their approach allows only one nested DFS procedure at a time.

6 Conclusion

We propose a simple distributed nested depth-first search algorithm for checking LTL properties in parallel environments. The algorithm assumes that the partition function localizes the cycles of the state space. No extra data structure or synchronization mechanism has to be added to the distributed algorithm and the search for accepting cycles can be done in parallel. Since all states of a strongly connected component belong to the same equivalence class and are explored in depth-first order, the algorithm remains correct.

We show that a static analysis of the model and the properties can be used in order to generate the desired partition functions. The structure of the never claim and the processes of the system are used to construct a partition such that states of a cycle belong to the same equivalence class.

Achieving scalability and a good load distribution are the main drawbacks of our approach. The number of classes that the partition function delivers depends on the structure of the model and the property to be checked. In the worst case no partition is possible and alternative methods like the one described in [10] has to be taken. Moreover, there is no way to know a-priori how the state sets are distributed among the equivalence classes. Therefore we propose our solution as simplified variant of a general algorithm to be applied in special cases.

References

1. D. Bader. A practical parallel algorithm for cycle detection in partitioned digraphs. Technical report, University of New Mexico, 1999.
2. S. Ben-David, T. Heyman, O.Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *Third International Conference on Formal methods in Vomputer-Aided Desing (FMCAD'00)*, November 2000.
3. B. Bollig, M. Leucker, and M. Weber. Parallel model checking for the alternation free μ-calculus. In *Proceedings of the 7th International TACAS Conference*, pages 543–558, 2001.
4. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
5. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering*, 1999.
6. L. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. In *IPDPS Workshops*, pages 505–511, 2000.
7. H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proceedings of the 8th International SPIN Workshop*, 2001.
8. W. Hendrickson, S. Plimpton, and L. Rauchwerger. Identifying strongly connected components in parallel. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Sci. Comput.*, 2001.
9. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In O. Grumberg, editor, *Computer Aided Verification, 12th International Conference*, volume 1855, pages 20–35. Springer-Verlag, 2000.
10. L. B. J. Barnat and J.Stbrn. Distributed ltl model-checking in spin. In *Proceedings of the 8th SPIN Workshop*, 2001.
11. F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. of the 5th International SPIN Workshop*, volume 1680 of *LNCS*. Springer-Verlag, 1999.
12. J. Reif. Depth-first search is inherently sequential. In *Information Processing Letters*, 1985.
13. U. Stern and D. L. Dill. Parallelizing the murphi verifier. In *Computer Aided Verification*, pages 256–278, 1997.