On the Sleep Sets Method for Partial Order Verification of Concurrent Systems

Maciej Koutny and Marta Pietkiewicz-Koutny
Department of Computing Science
University of Newcastle
Newcastle upon Tyne NE1 7RU, U.K.

Abstract

We discuss one of the versions of the 'sleep sets' method proposed to reduce the computational effort involved in state space search. We identify some of the problems in the algorithms presented in [2, 3] which use sleep sets to implement an efficient search of the state space of a concurrent system.

1 Introduction

The verification of concurrent systems based on state space exploration suffers from a problem which is usually referred to as the (combinatorial) *state space explosion.*[1] One of the approaches aimed at developing algorithms coping with it is the *sleep sets* method proposed and developed in [1, 2, 3, 4]. The approach is based on the partial order semantics of a concurrent system (more specifically the traces of [6]), and attempts to reduce the computational effort by ensuring that at most one interleaving of the partial order representing an abstract history of the system is ever generated during the search. In this report we look closely at one of the earlier versions of the sleep sets method presented in [2, 3]. We show that such a method may not generate the entire state space of a system, making it unsuitable for the verification purposes. We look at the possibility of modifying the algorithm of [3] which would be different from that presented in [4], and at the same time lead to a better computational result. We propose a mild modification to the algorithm of [3] and conjecture that it overcomes the problems of the latter. In the case of the algorithm introduced in [2], which combines sleep sets with state space caching, our conclusion is that it seems unlikely that a similar modification exists.

[1]Essentially, the size of the state space can be exponential in the size of a system.

2 Preliminaries

As in [3], we assume that a concurrent system P is composed of n ($n \geq 1$) concurrent processes, $P_1, \ldots, P_n$. Each P_i is represented by a finite automaton A_i, whereas P itself is represented by the sequence of automata,

$$A_P = (A_1, \ldots, A_n).$$

For every i, $1 \leq i \leq n$, A_i is defined as a tuple

$$A_i = (\Sigma_i, S_i, \Delta_i, init_i),$$

where Σ_i is an alphabet of actions, S_i is a finite set of states, $\Delta_i \subseteq S_i \times \Sigma_i \times S_i$ is a transition relation, and $init_i \in S_i$ is an initial state. We assume that for every $a \in \Sigma_i$ there is exactly one pair of states $s, r \in S_i$ such that $(s, a, r) \in \Delta_i$.[2] In what follows we will denote:

$$\begin{array}{lcll} \Sigma & = & \Sigma_1 \cup \cdots \cup \Sigma_n & \text{actions} \\ S & = & S_1 \times \cdots \times S_n & \text{states} \\ init & = & (init_1, \ldots, init_n) & \text{initial state} \end{array}$$

The behaviour of A_P is defined in terms of finite sequences of actions of Σ. It is assumed that each shared action has to be executed synchronously by all the automata whose action sets contain it.

Let $w = a_0 a_1 \ldots a_{k-1}$ be a sequence of actions of Σ. A_P *generates* w if there are states $s_0, s_1, \ldots, s_k$ of S such that $s_0 = init$ and for all i, $0 \leq i \leq k-1$,

$$s_i \xrightarrow{a_i} s_{i+1},$$

where $s_i \xrightarrow{a_i} s_{i+1}$ denotes a transition between states s_i and s_{i+1} through execution of action a_i, which is defined in the following way:
Let $s_i = (p_1, \ldots, p_n)$ and $s_{i+1} = (q_1, \ldots, q_n)$. Then $s_i \xrightarrow{a_i} s_{i+1}$ if for every $1 \leq j \leq n$,

$$\begin{array}{lcl} a_i \in \Sigma_j & \Rightarrow & (p_j, a_i, q_j) \in \Delta_j \\ a_i \notin \Sigma_j & \Rightarrow & p_j = q_j \end{array}$$

We then say that w is an *action sequence* of A_P *leading to* s_k, and that s_k is *reachable* from the initial state, *init*.

The semantics of A_P can be refined by taking into account the concurrency structure of the system. The resulting model, in which interleavings (action sequences) are replaced by partial orders, can be introduced in the form of Mazurkiewicz traces [6].

[2] Although the unicity condition is not assumed in [3], it simplifies the presentation without loss of generality.

```
Initialise: Stack is empty; H is empty;
Search() {
    enter init in H;
    push (init) onto Stack;
    }
    DFS();
DFS() {
        s=top(Stack);
        For all a enabled in s do {
            /* execution of a */
            s'=succ(s) after a
            if s' is NOT already in H then {
                enter s' in H;
                push (s') onto Stack;
                }
                DFS();
            /* backtracking of a*/
            }
        pop s from Stack
        }
```

Figure 1: Algorithm 1 - classical depth-first search

To define traces, we need the notion that two actions $a, b \in \Sigma$ are *independent*, $(a, b) \in ind$:

$$(a, b) \in ind \;\Leftrightarrow\; \forall 1 \leq i \leq n.\, a \notin \Sigma_i \vee b \notin \Sigma_i.$$

That is, two actions are independent if they belong to disjoint sets of automata constituting A_P. Traces are equivalence classes of action sequences. Two sequences belong to the same trace if one can be derived from the other by swapping (perhaps several times) adjacent independent actions. Formally, we define a relation $\sim$ on Σ^* such that $t \sim w$ if there are $t', w' \in \Sigma^*$ and $(a, b) \in ind$ such that: $t = t'abw'$ and $w = t'baw'$. The reflexive transitive closure of $\sim$ is denoted by $\approx$. The relation $\approx$ is an equivalence relation; its equivalence classes are called (Mazurkiewicz) *traces*. The trace containing a given action sequence t is denoted by $[t]$. It can be shown that each trace τ represents a partial order of action occurrences, and that the sequences in τ represent the possible linearisations of that partial order.

3 State Space Generation and Sleep Sets

Consider a classical depth-first search algorithm which can be used to generate the state space of A_P, i.e. all those states of S which are reachable from the initial state. Figure 1 shows its possible implementation taken from [2].

Algorithm 1 uses two main data structures: the *Stack* to store the states of the currently explored path from the initial state (root), and the hash table H to store the already visited states. For every reachable state of A_P it generates at least one action sequence leading to it. The state exploration method implemented by Algorithm 1 can be characterised as being 'on-the-fly', which means that there is no need to store transitions generated by the search, only the states encountered. Despite that, the actual memory requirements may still be very large. The sleep sets approach is a way of improving state space generation and other verification techniques; it has been proposed and used in, e.g., [1, 2, 3, 4]. We here focus on one of its earlier versions, used in [2, 3].

The method takes advantage of the distributed nature of the concurrent system represented by A_P and the ensuing partial order semantics based on Mazurkiewicz traces. The basic idea follows from the observation that starting from the initial state, all the action sequences of a given trace lead to the same state of the system. Therefore it is expected that the number of transitions explored during the search (or, in other words, the size of the search/transition tree) can be significantly reduced if at most one action sequence per trace be generated. This led to the introduction of the 'sleep sets' method. The way it works can be explained in the following way: Suppose that during the search we have reached a state s in which two independent actions, a and b, are enabled. It is clear that we can execute both ab and ba, and both these lead to the same state, s'. But this means that executing one of them is redundant. Suppose that a is executed before b in s. Then we may require that a is not to be executed in the state s'', where $s \xrightarrow{b} s''$. This still should be fine since executing a in s'' would anyway lead to the state s' reached after executing a followed by b. A formal device of preventing a from being executed in s'' is to put it into the associated 'sleep set' - a set of transitions which are enabled in a state yet one has decided to suspend them.

When modified to run with sleep sets, Algorithm 1 is transformed into Algorithm 2 (c.f. [2]), as shown in Figure 2.[3] In [2], it was stated that Algorithm 2 would visit all the reachable states of A_P. We have found that this is not the case, since it *may ignore some of the reachable states.* A possible counterexample, A_{P_1}, is shown in Figure 3. By inspection, one can see that Algorithm 2 can generate the transition tree shown in Figure 4.[4] We then observe that the state $s = (1,3,10,13)$ is reachable

[3]For a detailed explanation of the algorithm see [2, 3].

[4]In the diagrams, we indicate the order in which the nodes of a transition tree were generated and show, when appropriate, all non-empty sleep sets associated with the states.

```
Initialise: Stack is empty; H is empty;
Search() {
   enter init in H;
   push (init, ∅) onto Stack;
   }
   DFS();
DFS() {
      (s, Sleep)=top(Stack);
      For all a enabled in s and NOT in Sleep do {
         /* execution of a */
         s'=succ(s) after a
         /* computing sleep set of s' */
         Sleep' = {b | b ∈ Sleep ∧ (a, b) ∈ ind}
         if s' is NOT already in H then {
            enter s' in H;
            push (s', Sleep') onto Stack;
            DFS();
            }
         /* backtracking of a and adding it to the sleep set */
         /* if does not lead to a state on the stack */
         if s' is not in Stack then {
            Sleep = Sleep ∪ {a}
            }
         }
      pop (s, Sleep) from Stack
      }
```

Figure 2: Algorithm 2 - depth-first search with sleep sets

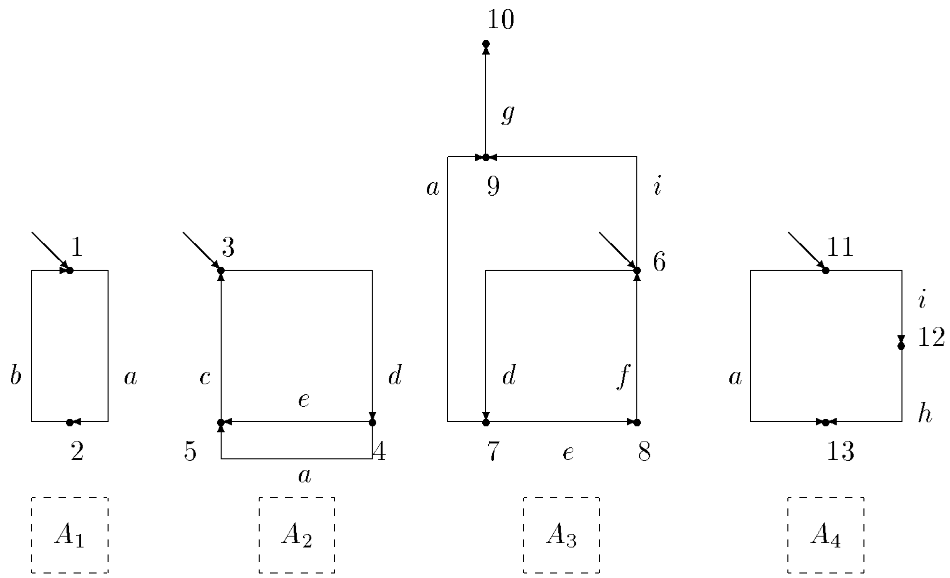

Figure 3: Example 1: $A_{P_1} = (A_1, A_2, A_3, A_4)$

in A_{P_1}, since we have the following sequence of transitions:

$$init = (1,3,6,11) \xrightarrow{i} (1,3,9,12) \xrightarrow{h} (1,3,9,13) \xrightarrow{g} (1,3,10,13) = s$$

yet Algorithm 2 does not visit it in Figure 4.

4 Discussion of Counterexample

To provide an additional insight into the problem identified by Example 1, we now discuss the technique used in [2] to prove Algorithm 2. As an example, we now consider A_{P_2} shown in Figure 5. (NB. A_{P_2} is not a counterexample for Algorithm 2.) The proof technique used in [2] (and also implicitly in [3]) was based on the following claim:

> “ Let Tr be the transition tree generated by Algorithm 1 (i.e. Algorithm 2 running without sleep sets), let Tr' be the transition tree generated by Algorithm 2, and let s be a reachable state. It is assumed that for both trees the execution order at the corresponding nodes (i.e. those labelled with the same state) of enabled transition is the same. Let π be the leftmost path in Tr leading to s. Then Algorithm 2 also generates π and thus s is visited.”

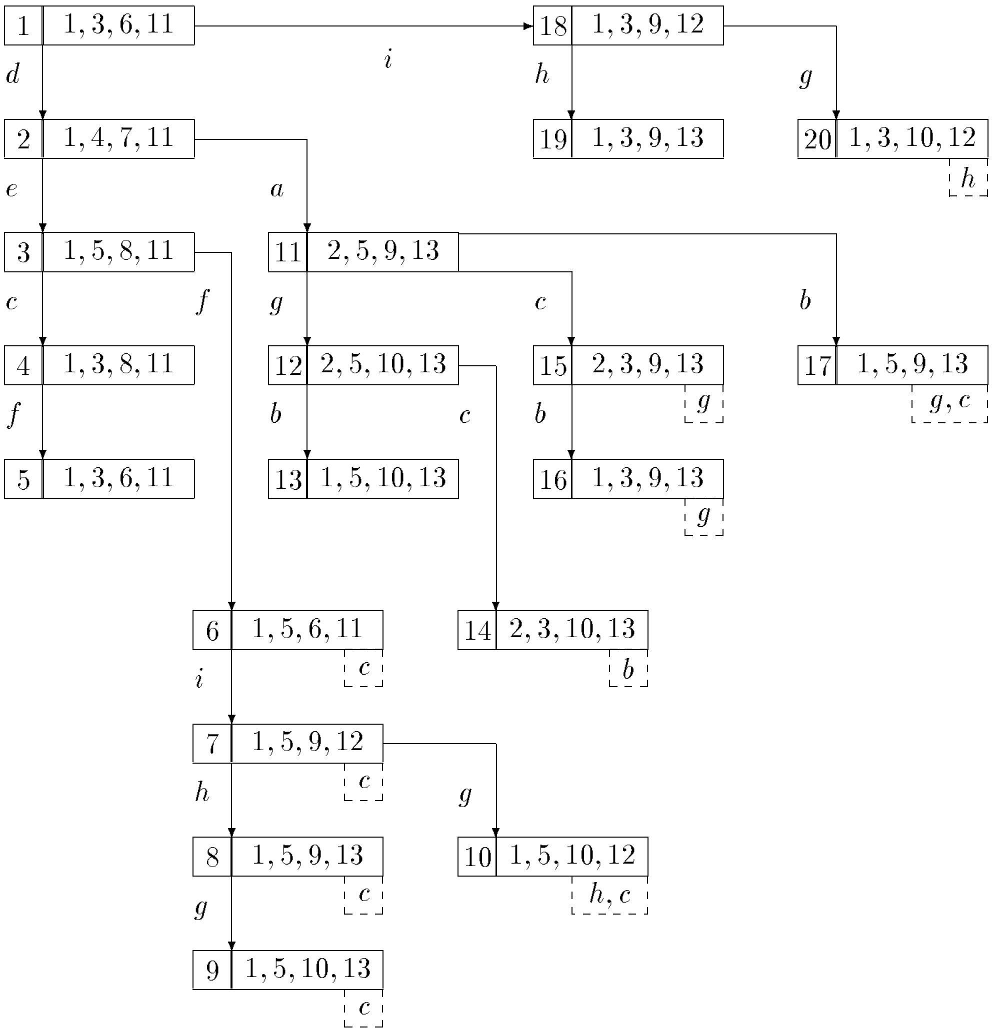

Figure 4: Transition tree generated by Algorithm 2 for Example 1

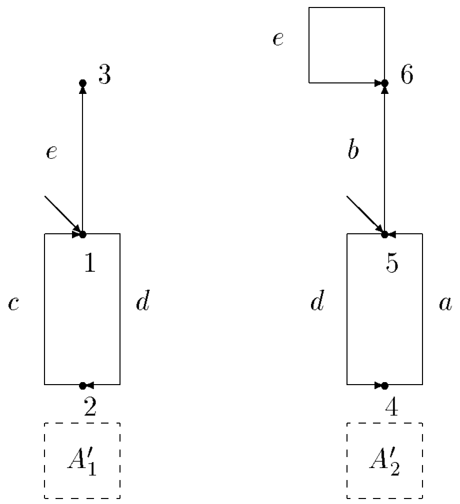

Figure 5: Example 2: $A_{P_2} = (A'_1, A'_2)$

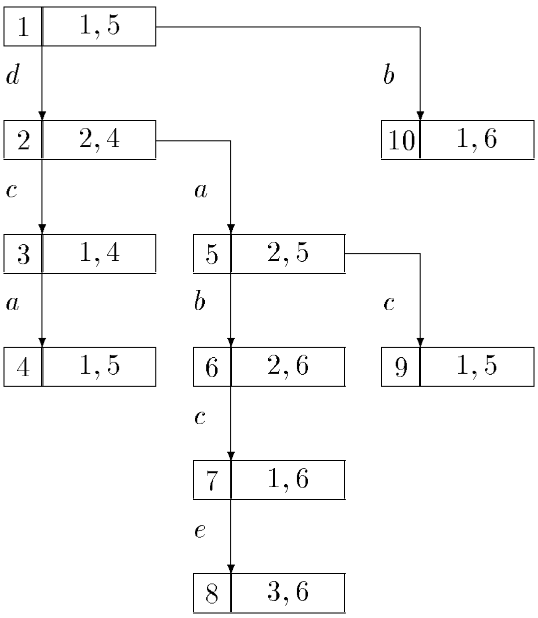

Figure 6: Transition tree Tr generated by Algorithm 1 for Example 2

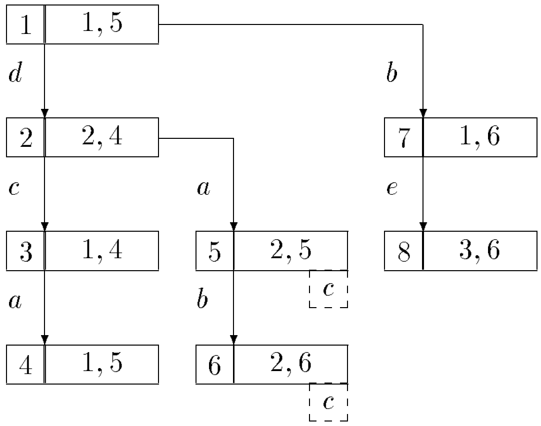

Figure 7: Transition tree Tr' generated by Algorithm 2 for Example 2

Example 2 shows that the above is not the case. In Figure 6 we show a possible transition tree generated by Algorithm 1, Tr, whereas in Figure 7, Tr' is shown. One may observe that the leftmost path $\pi = (1-2-5-6-7-8)$ in Tr leading to $s = (3, 6)$ is not contained in Tr'.

A question which now arises is whether Algorithm 2 could be modified in a way which would not destroy its intuitively appealing efficiency. By this we mean that there should be no changes to its overall structure, the actions executed in a reached state should be the same, and the storage requirements not bigger than in Algorithm 2. Under such conditions it seems that the only place where a modification could be made is the order in which enabled actions are selected for execution by the for-loop. It is also reasonable to require that any strategy used to order the actions there be static, otherwise the performance could be seriously degraded. In such a context, one can consider two possible ways of modifying Algorithm 2: The first is that one always changes the execution order of actions when visiting the next state in which they are enabled. That this would lead to an incorrect algorithm can be shown by taking Figure 4 and changing the order of the execution at node 7. The resulting transition tree, very similar to that in Figure 4, would again ignore the reachable state $s = (1, 3, 10, 13)$.

Another way of modifying Algorithm 2 would be to assume a fixed total ordering on actions which is always adhered to when one is about to execute the for-loop. We will call the resulting method Algorithm 2a.

Conjecture

Algorithm 2a generates all the reachable states of A_P.

Although we do not know yet whether the above property is true, it can be observed that it would require a proof technique different from that used in [2]. For in Figure 6 and 7 we assumed a fixed total ordering on actions:

$$d > b > c > a > e.$$

Quite different possibility of modifying Algorithm 2 would be to start comparing (and storing in H) pairs $(s, Sleep)$ rather than just states s. Indeed, the algorithm then becomes similar to Algorithm 3 described later in this report. It can also be proved correct in a similar way (using Theorem 6.1), but not using the proof technique from [2]. For in Figure 7 no state is generated more than once.

5 State Space Search with Caching

We now discuss the combination of the sleeps sets (Algorithm 2) and caching described in [2]. The method assumes that H has a limited capacity (usually smaller than the entire state space of the concurrent system) and when it becomes full, some states stored in H are deleted according to a pre-defined strategy (a number of these were discussed in [2]), to make room for newly generated states. Contrary to a statement in [2] that Algorithm 2 is suitable for state space caching, we will show that *Algorithm 2a (and thus Algorithm 2) with state space caching (implementing the FIFO policy) may ignore some of the states of the system.* A counterexample, A_3, is shown in Figure 8. When running Algorithm 2a for A_{P_3}, we assumed the following:

- H can hold up to 30 states.
- The caching policy is to remove the earliest state visited when H is full.
- The global ordering used for selecting enabled actions is:

$$h > g > j > b > c > d > i > e > a > f > a_1 > \cdots > a_{100}.$$

The transition tree generated by Algorithm 2a is shown in Figure 9. Again, the reachable state $s = (1, 3, 10, 13)$ was not visited. The crucial point is that when the algorithm generates node 116 (with the state $s' = (1, 5, 9, 13)$) the fact that s' has already been visited is no longer known as the hash table H does not contain s' inserted there when the node labelled with 8 was generated.

The above example together with Example 1, shows that both Algorithm 2 and Algorithm 2a cannot be used for efficient state space caching technique based on the FIFO policy.[5]

NB. We based the above discussion on the results presented in [2]. However, the examples can be suitably modified to apply to [3] as well.

[5]This does not mean, of course, that the same is true for all possible way caching can be implemented.

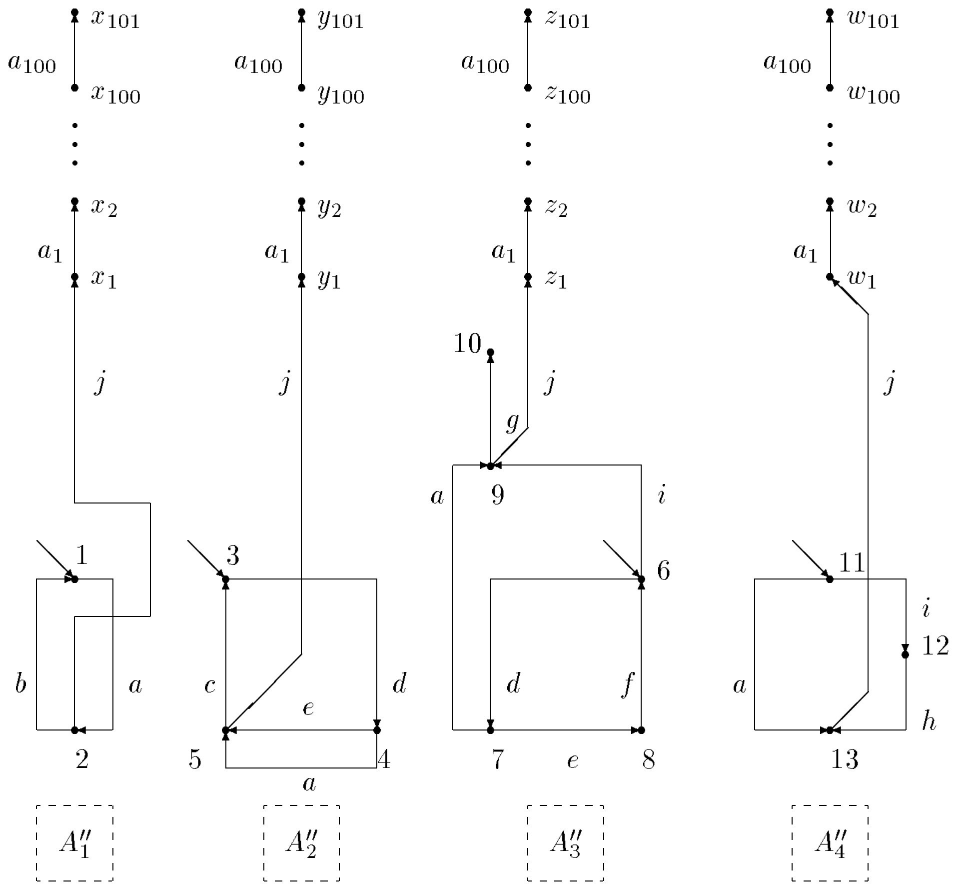

Figure 8: Example 3: $A_{P_3} = (A_1'', A_2'', A_3'', A_4'')$

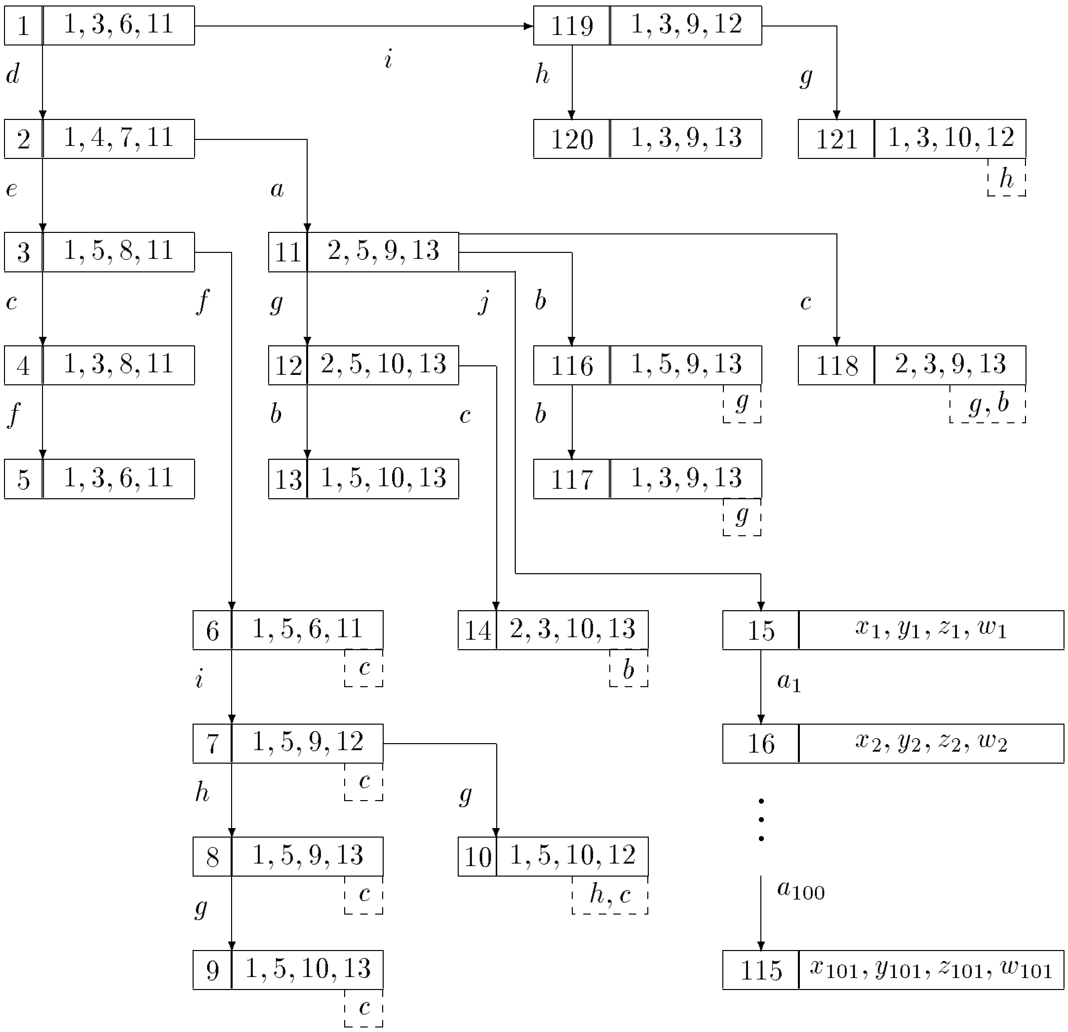

Figure 9: Transition tree generated by Algorithm 2a for Example 3

6 Alternative Sleep Sets Algorithm

In this section we discuss an algorithm based on the sleep sets, Algorithm 3, presented in Figure 10. It is similar to the version of the sleep sets method in [4]. This time not only the state reached is taken into account (and saved in the hash table H) when deciding to end the exploration of a given path from the root, but also the sleep set which has been generated.[6] In short, one can terminate the path if the sleep set currently reached is at least as big as the sleep set(s) with which the state was previously reached. In Algorithm 3, the hash table contains pairs $(s, SLEEP)$, where $SLEEP$ is a set of sleep sets for s. The stack, on the other hand, comprises pairs (s, R), where R is a single sleep set. We stress that the execution order for actions is not arbitrary in Algorithm 3. We do assume a *fixed ordering* on actions to be given at the outset which is then used for ordering enabled actions in every visited state.

Before proving that Algorithm 3 is correct, we prove an auxiliary result.

Theorem 6.1

Let T be a tree with labelled arcs, and let $\prec$ be a reflexive transitive relation on the nodes such that if $v \prec w$ and (v, a, v') is an arc outgoing from v then there is an arc (w, b, w') outgoing from w satisfying $a = b$ and $v' \prec w'$.
Let T' be a sub-tree of T (with the same root) such that each node in T' inherits either all or none of the successors nodes from T. Moreover, if v is a leaf node in T' but was not one in T, then there is v' in T' such that $v \prec v'$ and v' is not a leaf node in T'.

With the above assumptions, for every node v in T, there is a node v' in T' such that $v \prec v'$.

Proof: By induction. The base case is obvious. Suppose that (v, a, w) is an arc in T and there is v' in T' such that $v \prec v'$. If v' is not a leaf node then from the choice of T it follows that there is an arc (v', a, w') in T (and hence in T', by the choice of T') such that $w \prec w'$. If v' is a leaf node then, by the choice of T', there is v'' in T' which is not a leaf, such that $v' \prec v''$. From the transitivity of $\prec$ it follows that $v \prec v''$, and we proceed similarly as in the first case. ■ 6.1

[6] Intuitively, this means that the algorithm will in general generate more transitions than Algorithms 2 and 2a, and that it will need more storage space.

```
Initialise: Stack is empty; H is empty;
Search() {
    init.SLEEP = {∅};
    enter init in H;
    push (init, ∅) onto Stack;
    DFS();
    }
DFS() {
        (s, R)=top(Stack);
        For all a enabled in s and NOT in R do {
            /* execution of a */
            s'=succ(s) after a
            /* calculation of new sleep set */
            R' = {b ∈ R | (a, b) ∈ ind}
            if s' is NOT already in H then {
                s'.SLEEP = {R'};
                enter s' in H;
                push (s', R') onto Stack;
                DFS();
            }
            else
            if s' is in H but there is NO R'' in s'.SLEEP
                    such that R'' ⊆ R' then {
                s'.SLEEP = s'.SLEEP − {R'' | R' ⊆ R''} ∪ {R'};
                push (s', R') onto Stack;
                DFS();
                }
            /* backtracking of a*/
            R = R ∪ {a}
            }
        pop (s, R) from Stack
        }
```

Figure 10: Algorithm 3

Theorem 6.2

Algorithm 3 generates all the reachable states of the system in a finite number of steps.

Proof: The finiteness of Algorithm 3 follows from A_P being finite. The reachability result can be proved in the following way:

Let T be the tree defined inductively in the following way:

- the root is is labelled with $(init, \emptyset)$.
- Let (v, R) be a label of an already generated node, where v is a state and R is a sleep set for v (i.e. a set of actions enabled at v).
  Let $a_1, \ldots, a_k$ be actions enabled at v which are not in R and which are ordered according to the execution order assumed in Algorithm 3. We generate k new nodes labelled with

  $$(v_1, R_1), \ldots, (v_k, R_k)$$

  and k new arcs outgoing from the current node, labelled respectively with $a_1, \ldots, a_k$, and leading to these nodes, where for every i, $1 \leq i \leq k$,

  $$v \xrightarrow{a_i} v_i$$
  $$R_i = \{b \in R \mid (a_i, b) \in ind\} \cup \{a_j \mid j < i \wedge (a_i, a_j) \in ind\}.$$

- We define the relation $\prec$ on the nodes of the tree as follows: Let V be a node labelled with (v, R) and W be a node labelled with (w, R'). Then $V \prec W$ if $v = w$ and $R' \subseteq R$.

Clearly, $\prec$ is a reflexive transitive relation. We first need to show that T satisfies the condition from the formulation of Theorem 6.1. Suppose V is a node labelled with (v, R) and W is a node labelled with (v, R') and $V \prec W$, i.e. $R' \subseteq R$. Let $a_1, \ldots, a_k$ be the sequence of actions (ordered as above) labelling arcs outgoing from V, and similarly, $b_1, \ldots, b_l$ be the (ordered) sequence of actions labelling arcs outgoing from W. Consider the arc (V, a_s, V') where V' is labelled with (v', Q) satisfying the following:

$$v \xrightarrow{a_s} v'$$
$$Q = \{b \in R \mid (a_s, b) \in ind\} \cup \{a_i \mid i < s \wedge (a_s, a_i) \in ind\}.$$

Since $R' \subseteq R$ there is b_r such that $a_s = b_r$. Let Q' be such that (v', Q') is the label of the node W' to which there is an arc from W labelled with $b_r = a_s$. Let $j < r$ be such that $(b_j, b_r) \in ind$. Then either $b_j \in R$ or there is $i < s$ such that $b_j = a_i$ ($i < s$ follows from $j < r$ and the assumed fixed ordering of action selected for execution). In either case $b_j \in Q$ which implies that $Q' \subseteq Q$. Thus $V' \prec W'$.

Having shown that T is as in Theorem 6.1, we only need to prove that for each reachable state of the system, v, there is a node in T labelled with (v, R), for some sleep set R. This, however, can be shown similarly as Lemma A.2 in [3]. Then we apply Theorem 6.1. ■ 6.2

One can also show, using Theorem 6.1, that Algorithm 3 can be used in combination with state space caching yielding a method in which all reachable states are visited.

The basic advantage of Algorithm 3 is that it preserves the crucial property of Algorithm 2 in that it never generates two different execution paths belonging to the same Mazurkiewicz trace.

Theorem 6.3

If w and t are two different execution paths generated from the root by Algorithm 3 then $[w] \neq [t]$.

Proof: Similar as that of Theorem 3.2 in [2]. ■ 6.3

This theoretical result is further supported by an experimental comparison of Algorithm 2 and Algorithm 3. The table below shows the results of comparison of the two algorithms for the Round Robin Access Protocol described in [5] for a ring of two (rr2) to five (rr5) participants (the third and fourth columns show the number of transitions - arcs - in the respective trees):

rr$i$	No of states	Algorithm 2	Algorithm 3
rr2	18	25	25
rr3	54	78	74
rr4	144	204	196
rr5	360	500	488

Note that the trees generated by Algorithm 3 are 'smaller' than those generated Algorithm 2.[7] We have also carried out similar comparisons for other examples. Usually, the transition tree generated by Algorithm 3 were bigger than that for Algorithm 2, but never by more than by 7%.

[7]This supposedly paradoxical situation can be explained by the fact that generating a longer path can shorten a number of subsequent paths.

7 Concluding Remarks

We have discussed the version of the sleep sets method introduced in [2, 3] identifying problems within both the algorithms and the proof technique used. We have proposed a simple modification to one of those algorithms which as we conjecture overcomes the previous difficulties. We then discussed another modification (in line with that used in [4]) which is correct, but whose characteristics (in particular, in terms of the required storage space) is not as good as that of the former.

Acknowledgement

This work has been done within the Esprit Basic Research Working Group 6067 CALIBAN. We would like to thank Patrice Godefroid for helpful remarks.

References

[1] Godefroid P.: *Using Partial Orders to Improve Automatic Verification Methods.* In Proc. 2nd Workshop on Computer Aided Verification, Rutgers, June 1990.

[2] Godefroid P., Holzmann G.J. and Pirottin D.: *State Space Caching Revisited.* In Proc. 4th Workshop on Computer Aided Verification, Montreal, June 1992.

[3] Godefroid P. and Wolper P.: *Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties.* In Proc. 3rd Workshop on Computer Aided Verification, Aalborg, July 1991.

[4] Godefroid P. and Wolper P.: *Partial-order Methods for Temporal Verification.* Proc. Concur'93, E.Best (Ed.). LNCS 715, Springer 1993, 233-246.

[5] Graf S. and Steffen B.: In Proc. 2nd Workshop on Computer Aided Verification, Rutgers, June 1990. *Using Interface Specifications for Compositional Reduction.*

[6] Mazurkiewicz A.: *Trace Theory.* Lecture Notes in Computer Science, 279-324,1986.