Space Efficient Reachability Analysis Through Use of Pseudo-Root States

Atanas N. Parashkevov and Jay Yantchev

University of Adelaide, Adelaide, SA 5005, Australia

Abstract. This paper presents a novel reachability analysis technique which, while still maintaining a set of reached states, significantly reduces the size of this set through excluding a specific subset of those states, referred to as pseudo-root states. Pseudo-root states are states which are not reachable from the unexplored state space of the finite model. Such states may be safely discarded from state storage. The modified reachability analysis algorithm identifies and discards pseudo-root states at each iteration of the state search. For a set of three example problems, the presented algorithm results in 2 to 16 fold improvements in space requirements, while increasing the run time at most twice.

1 Introduction

Exhaustive exploration of a finite state model is a key part of most automatic verification algorithms, which rely on checking all reachable states of the model against a specification of desired properties. Conventional algorithms for traversing the model starting from its initial states require storing the set of all visited states. This is done in order to guarantee that each state is visited exactly once and thus avoid redundant work.

The need to store all reached states is a major performance bottleneck for automatic verification algorithms. The amount of memory required for storing these states is, in general, proportional to their number. It has been reported that use of virtual memory considerably slows down state matching [12, 8], thus the space available for state storage is effectively limited to the amount of physical memory. As the exploration involves a search through the stored states for each newly reached state, there is a growing time overhead as well. Several approaches have been proposed to alleviate this problem.

Symbolic model checking [4, 14] uses *Ordered Binary Decision Diagrams* (OBDDs) [1, 2] for encoding the finite model and the reached states. This brings several potential advantages. Firstly, the finite model is not constructed explicitly and, given a well chosen variable ordering, an OBDD consisting of a few thousand nodes may represent a model with 10^{30} states and more. Secondly, if the set of visited states exhibits some form of regularity, the OBDD representing this set may require many orders of magnitude less space than, for example, a hash table containing the corresponding state vectors. Thirdly, symbolic model checking allows sets of states, not only a single state, to be checked at each iteration of the reachability analysis procedure. Unfortunately, the size of OBDDs

representing the visited states is unpredictable, and in the worst case may be much larger than the sum of sizes of the corresponding state vectors.

Another reachability analysis method proposed in the literature is *scatter searching* [11]. The idea is to explore, within the limits imposed by the available memory or time, one or more randomly chosen subsets of the full finite model and then declare the system "error-free" with a certain degree of probability. An error found in any of these subsets is an error in the complete model. However, not finding any errors does not imply the complete correctness of the model. The *bit-state hashing* technique [12] enhances this method and allows scatter searching to be carried out considerably faster and in less space. However, this is essentially a probabilistic method, which may not be acceptable in cases where a definitive result is required.

State space caching approach [11, 13] stores in memory as many reached states as possible. After the available memory is filled up, newly generated states replace states from memory (chosen randomly or otherwise), the latter serving as a cache of the set of visited states. As states discarded from memory may be reached again at a later stage of the reachability analysis, there is a considerable risk of doing redundant work. In practice, using a cache of size less than a third of all reachable states brings unacceptable time penalties [11, 8]. An enhancement of the state space caching technique [8] uses *partial ordering* methods [7, 19, 18] to deal with interleavings of independent transitions and reduce the number of times a state is visited during reachability analysis. However, a state may be reachable more than once for reasons other than pure interleaving — one example being cyclic behavior. Thus, the risk of revisiting discarded states, although reduced, is still present.

More recently, an approach combining state space reduction, partial order driven reachability analysis, and a modification of the bit-state hashing technique has been proposed [15]. A precomputation stage involving exhaustive state-space exploration of the finite model is used to gather approximate information about the number of times each state of the model is to be revisited during finite model exploration. This information is then utilized in the course of the actual model checking for a more selective removal of states from memory when space limits are reached as compared to that used in [8].

With the exception of the refined state space caching technique from [8] and its further improvement suggested in [15], a common feature of the existing reachability analysis methods is that states are treated as separate entities outside of the finite model they belong to. In other words, the potentially advantageous information about states and transitions in the model is not utilized; states are stored obliviously.

This paper presents a different approach to the state space storage problem. Whereas the refined state space caching technique extracts the necessary information by applying higher-level syntactic conditions on the system description, our method is lower-level in that it works by utilizing information at the level of the model itself. More specifically, our approach keeps track of the visited predecessors for each state and as soon as all predecessors of a state are ex-

plored it is discarded from memory. States that are no longer reachable from the unexplored state space are referred to as *pseudo-root states*. We present an efficient algorithm for the identification and deletion of a visited state that becomes pseudo-root. Complexity analysis shows that this algorithm incurs at most a two-fold increase of the runtime compared to the conventional reachability analysis algorithm. Our experimental results indicate that our technique allows exhaustive state space exploration visiting each state *exactly once* while storing, at any one time, between 6–45% of the reachable state space.

The rest of this paper is organized as follows. Section 2 considers a generic version of the conventional reachability analysis algorithm, and presents two examples which illustrate the intuitions behind our approach. Section 3 defines the notion of pseudo-root states and presents the modified state space exploration algorithm. Then the time and space complexity of our algorithm is analyzed. Section 4 comments on the experimental results obtained with a modified version of our tool for automatic verification ARC [17]. In Section 5 we discuss issues concerning the practical application of the pseudo-root state method. Our conclusions are given in Section 6.

2 Background and Motivation

2.1 Labeled Transition Systems

As a suitable representation of finite models we use *Labeled Transition Systems* (LTS). A finite nondeterministic LTS $\mathcal{L}$ is considered to be a tuple:

$$\mathcal{L} = \langle S, A, R, I \rangle$$

where S is the finite set of states in the LTS; A is the finite set of actions (events) in the LTS; $R \subseteq S \times A \times S$ is the set of labeled transitions in the LTS; $I \subseteq S$ is the set of initial states of the LTS.

If $\sigma \in S$ is a state in the LTS $\mathcal{L}$, the following functions are defined:

$$\mathsf{pred}(\sigma, \mathcal{L}) = \{\theta \mid \exists\, a \in A : \theta \xrightarrow{a} \sigma \in R\}$$
$$\mathsf{succ}(\sigma, \mathcal{L}) = \{\theta \mid \exists\, a \in A : \sigma \xrightarrow{a} \theta \in R\}$$

Function pred computes the set of all states from which σ can be reached via a single transition (predecessor states), while function succ derives the set of all states reachable from σ via a single transition (successor states).

2.2 Conventional Reachability Analysis

A generic version of the conventional reachability analysis algorithm performing exhaustive state space exploration is presented in Fig. 1. Let us call it Algorithm 1. Two sets of states are maintained, checked for the states that have been visited, and pending for the states that have been reached but not checked yet. Throughout this paper we use interchangeably the terms *visited*, *explored* or

```
procedure Explore(L: in LTS) is
  s, r: State;
  checked, pending: set of State;
begin
  checked := {};
  pending := I;
  while pending ≠ {} loop
    select s ∈ pending;
    pending := pending − {s};
    -- check(s);
    checked := checked ∪ {s};
    for r ∈ succ(s,L) loop
      if r ∉ checked ∪ pending then
        pending := pending ∪ {r};
      end if;
    end loop;
  end loop;
end Explore;
```

Fig. 1. Algorithm 1: Conventional reachability analysis

checked for states from the set checked, and refer to states from the set pending as *pending*. We call the states in checked ∪ pending *reached*, and the rest of the states in S *unreached*.

Initially, checked is empty, and pending contains the set of initial states of $\mathcal{L}$. Algorithm 1 is iterative and terminates when pending becomes empty. Each iteration explores one state s selected from pending, which is then removed from pending and added to checked. The optional procedure check performs a generic verification check of the current state s. At the end of each iteration the successor states of s are examined and those not reached yet are added to pending. Termination is assured by the monotonic growth of checked and the finiteness of S — eventually all pending states will have to be checked.

Two strategies for selecting a state from pending have become most popular. Breadth-first search (BFS) implements the set pending as a FIFO queue (list), while depth-first search (DFS) stores the states from pending in a LIFO queue (stack)[1].

It may be observed that the set checked is used solely for determining if the successor states of s have been reached already, ensuring that each state is visited exactly once. However, some states in checked may be unreachable from outside of the explored state space. Keeping such states in checked is unnecessary and also undesirable because that brings about performance penalties:

– These states take up valuable space;

[1] The original DFS strategy actually requires slight modification of Algorithm 1, which is, however, left out for the sake of clarity and conciseness of our presentation.

– The computation of the condition "r $\notin$ checked $\cup$ pending" is slower due to the larger size of checked.

In the rest of Section 2 we present two simple examples which illustrate that some states in checked need not be kept there throughout the search. These examples serve as motivation and encouragement for this work.

2.3 Interleaved Processes Example

Consider two very simple models P and Q combined by interleaving (Fig. 2) and assume that the product LTS is explored by Algorithm 1 performing BFS. The search starts from the initial state 0, which is then put in checked while its successor states 1 and 2 are put into pending. However, state 0 is a root state (a state with no incoming transitions) in the LTS, and therefore it is not reachable from any other state in that LTS. It may be safely discarded from checked without affecting the future behavior of Algorithm 1.

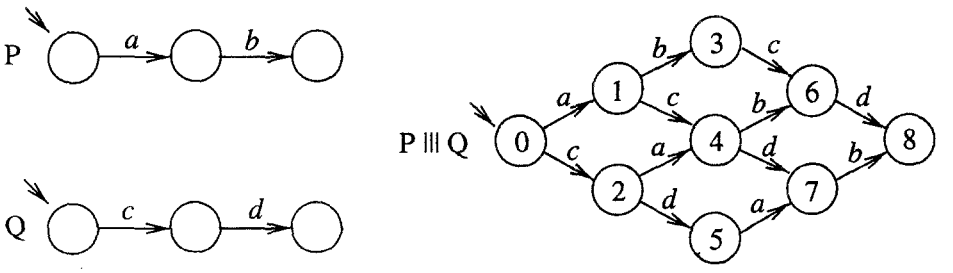

Fig. 2. LTS of the interleaved processes example

At the next iteration state 1 is explored and moved into checked. States 3 and 4 are put into pending. State 1 may only be reached from state 0, which itself has been already identified as unreachable. Therefore, state 1 is no longer reachable, too, and its removal from checked will not affect the rest of the LTS exploration. A similar conclusion may be drawn for state 2 as well.

Following this line of reasoning at every step until the end of the reachability analysis, it becomes evident that the LTS can be exhaustively explored, visiting each of the 9 states exactly once, by storing no more than 3 states (all in pending and none in checked) at any one time. All that is required is extracting and utilizing some information about the states from the LTS.

2.4 Counting Example

This example models the behavior of a finite counter. Figure 3 presents the LTS of a counter between 0 and 3. Each of the states is labeled with the value of the

counter, which may be incremented by an *inc* transition and decremented by a *dec* transition. Again, let us assume that the LTS of this example is explored by Algorithm 1 using BFS.

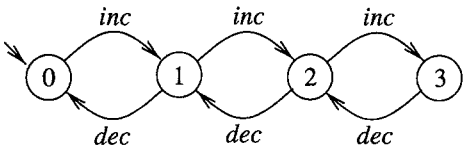

Fig. 3. LTS of the counting example

Reachability analysis starts from the initial state 0. State 0 is moved to checked and its only successor state 1 is put into pending. At the next iteration, state 1 is explored and put into checked, while its successor state 2 is stored in pending. At this point, checked consists of states 0 and 1. However, any path in the LTS starting from an unreached state and ending at state 0 must contain state 1, as state 1 is the only predecessor of state 0. Therefore, state 0 may be discarded from checked without any risk of revisiting it again.

When state 2 is explored, it is moved into checked and state 3 is put into pending. With states 1 and 2 in checked, state 1 may only be reached through state 2 similarly to the previous iteration. State 1 need not be kept in checked and may be discarded. State 3 is the last to be explored, as state 2 is its only successor state, pending becomes empty and reachability analysis terminates.

Although we have used a counter from 0 to 3 as a particular example, it is not hard to see that any finite counter can be exhaustively explored while storing only two of its states simultaneously. As the exploration progresses, we are able to minimize the number of states in checked by leaving in only a subset of the reached states which, in a way, still uniquely represents (identifies) the full set of reached states.

3 Our Approach

3.1 Pseudo-root States

The informal reasoning used in Sections 2.3 and 2.4 is based on two propositions for safely discarding an explored state. Suitable candidates for discarding from checked are either root states (which can obviously be only members of I) or states reachable only through other visited states. We call a state of the latter kind a *pseudo-root state.*

Definition 1. A state σ in an LTS $\mathcal{L}$ is a pseudo-root state with respect to a set of states $\Sigma \subseteq S$ iff Σ contains all predecessors of σ, e.g. $\mathsf{pred}(\sigma, \mathcal{L}) \subseteq \Sigma$.

The interesting case is when Σ in the above definition is the set checked from Algorithm 1, and in the rest of this paper, when we write pseudo-root state we mean pseudo-root state w.r.t. checked. Since checked grows monotonically from the empty set to the set of all reachable states, once a visited state becomes pseudo-root, it remains pseudo-root throughout the rest of the search. By definition a pseudo-root state is unreachable from any state which is not in checked, and due to the properties of Algorithm 1 discussed in Section 2.2, it does not affect the rest of the search. Therefore, a state can be safely deleted from checked as soon as it becomes pseudo-root. To implement this, a technique for identifying pseudo-root states is required.

3.2 Modified Reachability Analysis Algorithm

A state becomes pseudo-root as soon as all of its predecessors have been visited. A straightforward implementation of this rule would be to keep, for every reached state, the set of its visited predecessors and upon each update of that set compare the latter with the set of all predecessors of the state. Although feasible, such a solution would be very ineffective, because the space and time overheads for keeping sets of states for each reached state are substantial.

Fortunately, there is a simpler way of identifying pseudo-root states. Assuming that memory limits are not hit, Algorithm 1 has the property of exploring each state in the LTS exactly once (although a state may be reached more than once). Therefore, the set of its predecessor states need not be kept in memory or recomputed at each iteration of the search — a simple counter of the number of predecessors yet to be visited would suffice. This idea is similar to the *reference counting* approach used for garbage collection of dynamically allocated memory [5, 9].

An implementation of the above pseudo-root state detection method is given in Fig. 4. Our modified reachability analysis algorithm, referred to in the rest of this paper as Algorithm 2, is based on Algorithm 1. Every state stored in checked or pending is paired with an integer number counter equal to the number of times that state is to be reached again. At the beginning, checked is empty and the initial states of $\mathcal{L}$ are put into pending. At each iteration of the main loop, a state s is selected for exploration and removed from pending. If the reference count of s is not zero then s may be reached again through a predecessor which has not been visited itself yet, and thus s is put into checked. If the reference count of s is zero then s has become pseudo-root and therefore can be discarded without ever been put in checked.

Next, all successor states r of s are considered. If r is a member of checked, then its reference count is at least one. If the reference count is equal to one, then s is the last of all predecessors of r which identifies r as pseudo-root and r is discarded from checked. If the reference count of r is greater than one, it is decremented by one to reflect the fact that another of its predecessors has been visited. In case r is a member of pending, its reference count is also decremented by one but the state is not discarded even if the result is zero, because a pending state has yet to be visited. If r has not been reached yet, it is included in pending paired

```
procedure ExplorePRS(L: in LTS) is
  s, r: State;
  counter: Array (State) of Integer;
  checked, pending: set of State;
begin
  checked := {};
  pending := I;
  for s ∈ I loop
    counter(s) := #pred(s,L);
  end loop;
  while pending ≠ {} loop
    select s ∈ pending;
    pending := pending − {s};
    -- check(s);
    if counter(s) > 0 then
      checked := checked ∪ {s};
    end if;
    for r ∈ succ(s,L) loop
      if r ∈ checked then
        if counter(r) = 1 then
          checked := checked − {r};
        else
          counter(r) := counter(r) − 1;
        end if;
      elsif r ∈ pending then
        counter(r) := counter(r) − 1;
      else
        pending := pending ∪ {r};
        counter(r) := #pred(r,L) − 1;
      end if;
    end loop;
  end loop;
end ExplorePRS;
```

Fig. 4. Algorithm 2: Modified reachability analysis

with its number of predecessors decreased by one (as s, one of r's predecessors, is already a checked state).

3.3 Reaching Memory Limits

Algorithm 2 as presented in the previous Section does not consider the possibility of exhausting the available memory. An interesting question is what can be done when, despite all efforts, the size of **checked** ∪ **pending** exceeds the available space. If the algorithm is to complete successfully then states which are *certain* to be reached again have to be discarded.

One option is to discard a pending state with a reference count greater than zero. Such a state is certain to be reached again and therefore will not be missed

if deleted from **pending**. This, of course, does not solve the space shortage problem, but rather provides means for a graceful degradation of the performance of Algorithm 2.

At some stage, a checked state which is not pseudo-root may have to be discarded. State space exploration may "leak" back into the reached state space and visit a state more than once. Under these conditions DFS strategy is required in order to guarantee termination. Essentially, our algorithm starts to approximate the state space caching algorithm with DFS presented in [11, 13]. It should be noted, however, that Algorithm 2 is expected to reach the memory limits much later in the search than the other algorithms because of the slower growth of **checked**.

3.4 Complexity Analysis

As with any other approach addressing the state space explosion problem, the performance of Algorithm 2 greatly depends on the actual example it is applied to, and more specifically, on the properties of the LTS graph of the model. An in-depth complexity analysis should therefore consist of not only worst-case, but also best-case and, if possible, average-case scenario performance. To estimate space and time complexity of the proposed algorithm we present a relative comparison with Algorithm 1.

Space complexity. As Algorithm 1 stores all reached states in $\mathcal{L}$, its space requirements are of the order of $|S| \times K$, where K is the size of a single state in bits. Note that $K \geq \lceil \log_2 |S| \rceil$. Algorithm 2 is in general expected to store far fewer reached states, but for each of these states requires additional space for storing a reference counter. The size of the reference counter must be at least $\lceil \log_2(d_{\max}) \rceil$ bits, where $d_{\max}$ is the maximum number of predecessors for a state in the LTS.

The worst-case space requirements of Algorithm 2 can be expected when $d_{\max} = |S|$, that is, when there is a state in S which has incoming transitions from all states in S. Assuming that the pseudo-root state method cannot discard any states from **checked** until the end of the search, and that $K = \lceil \log_2 |S| \rceil$, we obtain that the worst-case space complexity of Algorithm 2 is twice as much as that of Algorithm 1.

Algorithm 2 would perform best on a model in which every state has at most one predecessor — much like the counting example from Section 2.4, but without the *dec* transitions. Then one bit would suffice for the reference count, and Algorithm 2 would store at most one state at any one time. Such an LTS can be explored using only $K + 1$ bits of storage.

For most practical examples we have come across so far each state has at most several incoming transitions and thus a reference counter of only a few bits length is sufficient. The state vector of such examples may require hundreds of bits [12, 7]. Therefore, the average increase in space requirements for a single state is negligible. On the other hand, experimental results presented in Section 4

show that Algorithm 2 needs to store at any one time between 2 to 16 times fewer states than Algorithm 1. This far out-weighs the additional storage required by the reference counters.

Time complexity. In general, both algorithms execute the same number of iterations, exploring each state exactly once and following each transition exactly once. However, Algorithm 2 performs two extra computations as compared to Algorithm 1. Firstly, it sets the reference counter of every reached state by computing the set of its predecessor states, and secondly, it decrements this counter for every reached predecessor. The total number of decrements is less than or equal to the number of transitions in the LTS, because a state is connected to each of its predecessors by one or more unique transitions. As decrement and comparison operations are much faster (and take constant time) compared to state space search, we may ignore them in the analysis. The important factor in the analysis is the computation of predecessor states.

In a given LTS $\mathcal{L}$, the total number of successor states is equal to the total number of predecessor states. If we assume that computing predecessor states is of the same computational cost as deriving successor states then, in worst case, an iteration of Algorithm 2 would require approximately twice as much time as an iteration of Algorithm 1 does. For both algorithms the number of iterations is equal to the number of reachable states in S. Therefore we may conclude that the worst-case running time of Algorithm 2 is double that of Algorithm 1.

This informal worst-case analysis ignores the time required to search through reached states. To estimate average and best-case time performance of Algorithm 2 one would require some knowledge of the time complexity of state matching relative to the complexity of computing a successor state, which is implementation dependent (e.g. the LTS may be in OBDD form or computed on-the-fly, states may be stored in a hash table or in a balanced tree, etc.). In ARC, for example, the search through the reached states takes only a small fraction of the total run-time and any improvements in state matching brought about by the exclusion of pseudo-root states from the search set have only a marginal effect on the run-time. In tools like SPIN, where state matching takes up a substantial portion of the total run-time [12], the average and best-case time performance of Algorithm 2 may be considerably better than worst-case.

4 Experimental Results

In this section we present some early experimental results for the pseudo-root state based reachability analysis algorithm. We have modified the ARC tool [17] to implement Algorithm 2 with BFS. ARC is a formal verification tool based on the CSP algebra [10] and exploiting OBDD based techniques for working with LTS and sets of states.

4.1 Dining Philosophers

Our first example is the well known dining philosophers problem from [10]. A system consisting of n philosophers ($n \geq 2$) and n forks has to be checked for possible deadlocks. This example is indeed known to contain a deadlocked state, but in the context of this paper we are more interested in the performance of Algorithm 2. Table 1 summarizes the experimental data obtained.

Number of philosophers	Number of states	Peak number of states stored by Algorithm 2	Percentage of total states
3	154	69	44.8%
4	832	370	44.5%
5	4474	1940	43.4%
6	24040	10171	42.3%

Table 1. Experimental results for dining philosophers example

The space reductions offered by Algorithm 2 for this example are approximately two-fold and are in fact the lowest of all examples we have run. We attribute this to the cyclic behavior of both philosophers and forks and the form of the LTS of this example, which has bisectional width several times larger than its depth. It can be expected that DFS would be more suitable than BFS for such LTS, and we plan to experiment with it in the near future.

4.2 Milner's Schedulers

Milner's schedulers problem from [16] is another widely used "benchmark" example for verification tools. The example consists of n processes organized in a ring that have to be scheduled in sequence. Again, we compare the peak number of states stored by Algorithm 2 to the total number of reachable states (Table 2).

Number of processes	Reachable states	Peak number of states stored by Algorithm 2	Percentage of total states
3	33	11	33.3%
5	193	50	25.9%
7	1025	206	20.0%
9	5121	816	15.9%
11	24577	3262	13.3%

Table 2. Experimental results for Milner's schedulers

Perhaps the most interesting observation that can be made from Table 2 is that space improvements grow with the number of schedulers. This can be explained by the slower growth of the bisectional width of the LTS of this example as opposed to the growth of the number of states.

4.3 Monkey Puzzle

Monkey puzzle is a game by Holger Schemel played on a board of 4x5 square cells (Fig. 5). It is a variant of the popular 15 piece game on a square 4x4 board. The monkey puzzle contains several pieces of unequal size, including a 2x2 piece called "monkey". The objective of this game is to put the "monkey" on its "feet" (Fig. 5b) starting from the initial placement of pieces as shown in Fig. 5a by sliding the pieces on the board.

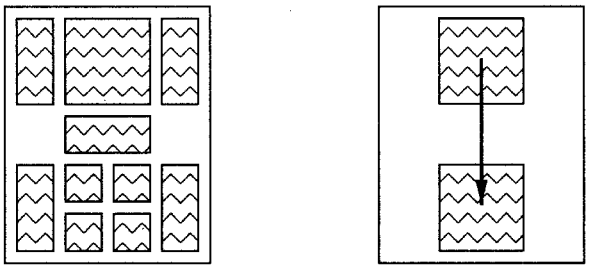

Fig. 5. The monkey puzzle

Solving the puzzle automatically involves exhaustive state-space search and can be therefore carried out by a verification tool given a proper description of the puzzle. We have adapted a description of the problem in CSP originally written by Bill Roscoe. Unlike the other examples presented, monkey puzzle has a very large state space and thus we have modified Algorithm 2 to terminate immediately after the first solution has been found. In our test run 23,679 states of the puzzle have been explored before finding a solution to it, whereas only a maximum of 1,425 states have been stored at any one time. In other words, with this example Algorithm 2 results in more than 16 fold reduction in memory requirements.

5 Discussion and Future Work

5.1 Computing Predecessor States

As it was stated in Section 3.4, the time complexity of the proposed pseudo-root state based reachability analysis (Algorithm 2) is dependent on the ability to

effectively compute the number of predecessor states.

When the full LTS is available in OBDD form (as is the case with the ARC tool [17]) computing predecessors poses no significant problems. In this case, having the complete transition relation of the finite model allows the computation of the predecessor states to be done using the same *relational product* operation proposed for the derivation of successor states [4, 3].

If "on-the-fly" verification technique is used, computing predecessor states might be slightly more complicated, but not necessarily less efficient. For example, the FDR2 tool [6] stores the global LTS implicitly by storing the LTSs of the sequential components and applying the synchronization rules to infer the complete operational behavior including successor states. Computing predecessors in this setup appears to be not too different from the problem of computing successors and to require comparable time.

5.2 BFS, DFS and Alternatives

Although our prototype implementation of Algorithm 2 uses BFS as the underlying search strategy, BFS may not be optimal for every LTS. One can expect that for reachability graphs whose bisection width is greater than their depth, DFS would tend to offer better space savings. In general, neither BFS nor DFS can be expected to provide the best space savings for all cases. An open question for future exploration is whether there exists a heuristic criteria for dynamically selecting a state from pending that can be used to maximize the space benefits of discarding pseudo-roots from the set of stored states.

5.3 Comparison to previous work

The main difference between our approach and the refined state space caching technique from [8] is in the type of information about states that is used by the algorithm and the level at which it is obtained. Algorithm 2 works by utilizing information about the predecessors of a state extracted from the LTS, whereas the sleep set method presented in [8] makes use of information about transition dependencies extracted at the higher syntactic level of system description (the source). Our method is independent of the input language and its semantics and is therefore applicable to a wide range of tools. Also, it allows space reduction for tightly coupled concurrent systems such as the monkey puzzle, for which a partial ordering technique would have little or no effect.

The approach discussed in [15] relies on *approximate* information on each state predecessors gathered in the preprocessing of the finite model. A hash function is used to map states of the model onto a (smaller) number of statically allocated predecessor counters. The total number of predecessors for all states that map to a particular counter is calculated and stored in that counter during the preprocessing stage. When the actual model checking is performed, a state can be safely discarded only when its corresponding counter becomes zero. This technique can be seen as a very coarse approximation of the pseudo-root state approach, which associates a predecessor counter with each state and thus

provides *exact* information on state predecessors. With our approach states are discarded as soon as they become pseudo-root, whereas the technique described in [15] requires that a set of states, all mapped to the same counter, become pseudo-root in order to safely discard them. An additional advantage of the pseudo-root state technique is that predecessor counters are allocated and discarded dynamically with their corresponding states, instead of being statically allocated.

We believe that it is possible to combine the benefits of the pseudo-root state and partial ordering methods, thus deriving a technique resulting in a better space reduction than would be possible by either one alone. Indeed, with such a combined approach it is to be expected that both fewer transitions will be traversed and also that states will be identified earlier as pseudo-root. This is because transitions not traversed by the sleep method can also be excluded from the computation of a state's predecessors.

6 Conclusions

We have presented an improved reachability analysis technique that works by identifying states that may be safely discarded from the set of stored states during analysis. Our experimental results show significant and promising improvements in space requirements. Existing tools should need little modification in order to incorporate the proposed method, mainly to implement the computation of predecessor states. We envisage further improvements in memory efficiency by combining the pseudo-root state technique with existing partial ordering methods.

Acknowledgments

This research has been partially supported by the Australian Department of Employment, Education and Training. The authors would like to thank the anonymous referees for their helpful and encouraging comments.

References

1. R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Trans. Computers*, C-35(8):677–691, August 1986.
2. R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
3. J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.
4. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

5. J. Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13:341–367, 1981.
6. Formal Systems (Europe) Ltd. *FDR2 User Manual*, August 1996.
7. P. Godefroid. *Partial-order methods for the verification of concurrent systems.* PhD thesis, University of Liège, Belgium, 1994.
8. P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *CAV '92*, pages 178–191. Lecture Notes in Computer Science, 663, Springer Verlag, 1992.
9. P. H. Hartel. Comparison of three garbage collection algorithms. *Structured Programming*, 11(3):117–128, 1990.
10. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
11. G. J. Holzmann. Automated protocol validation in Argos: Assertion proving and scatter searching. *IEEE Trans. on Software Engineering*, 13(6):683–696, 1987.
12. G. J. Holzmann. An improved protocol reachability analysis technique. *Software — Practice and Experience*, 18(2):137–161, 1988.
13. C. Jard and T. Jéron. Bounded-memory algorithms for verification on-the-fly. In *CAV '91*, pages 192–202. Lecture Notes in Computer Science, 575, Springer Verlag, 1991.
14. K. L. McMillan. *Symbolic Model Checking — an approach to the state explosion problem.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.
15. H. Miller and S. Katz. Saving space by fully exploiting invisible transitions. In *CAV '96*, pages 336–347. Lecture Notes in Computer Science, 1102, Springer Verlag, 1996.
16. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
17. A. N. Parashkevov and J. Yantchev. ARC — a tool for efficient refinement and equivalence checking for CSP. In *IEEE Int. Conf. on Algorithms and Architectures for Parallel Processing ICA3PP '96*, pages 68–75, 1996.
18. D. Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94*, pages 377–390. Lecture Notes in Computer Science, 818, Springer Verlag, 1994.
19. A. Valmari. On-the-fly verification with stubborn sets. In *CAV '93*, pages 397–408. Lecture Notes in Computer Science, 697, Springer Verlag, 1993.