observations lead us to the discovery of the incompatibility of the nested depth first search algorithms with partial order reduction.

References

[1] C. Courcoubetis, M. Vardi, P. Wolper, M, Yannakakis, Memory-efficient algorithms for the verification of temporal properties, Formal methods in system design 1 (1992) 275–288.

[2] P. Godefroid, G.J. Holzmann, On the Verification of Temporal Properties In Proc. PSTV93,, Protocol Specification Testing and Verification, Liege, Belgium, 1993, North-Holland, 109–124.

[3] G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall Software Series, 1992.

[4] G.J. Holzmann, D. Peled, An improvement in formal verification, *7th International Conference on Formal Description Techniques*, Berne, Switzerland, 1994, 177–194.

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

[6] M.Y. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification, *1st Annual IEEE Symposium on Logic in Computer Science*, 1986, Cambridge, England, 322–331.

```
never { /* non-progress: $\Diamond\Box\neg progress$ */
    do
    :: skip
    :: !progress -> break
    od;
accept: do
    :: !progress
    od
}
```

Figure 6: Non-Progress Büchi Automaton, expressed as Never Claim

Although we do not know how to modify the algorithm from Figure 5 for compatibility with partial order reduction, there is a simple alternative. The non-progress property can trivially be expressed as a model checking problem, for instance by expressing it with the LTL property: $\Diamond\Box\neg progress$, where $progress$ is a boolean function that yields $true$ in progress states, and $false$ otherwise. This LTL property can be translated into the Büchi automaton shown in Figure 6, which can be used to solve the problem with the algorithm from Figure 4.

The method from Figures 4 and Figure 6 are part of the latest version of the model checker SPIN (Version 2.9). The automaton from Figure 6 does not have to be specified by the user, but is automatically generated by SPIN when the user selects a search for non-progress cycles. The addition of the two-state non-progress automaton in Figure 6 can in the worst case double the memory requirements of the search based on Figure 4, compared to a direct application of the algorithm from Figure 5 (avoiding the need for the automaton from Figure 6). The algorithm from Figure 4, however, can be optimized significantly by combining it with a partial order reduction strategy. The measurements in [4] suggest that a reduced search based on Figure 4 should in almost all cases outperform an exhaustive search based on Figure 5.

7 Conclusion

The original algorithms for performing nested depth first searches to detect the presence of acceptance cycles ([1]) or the presence of non-progress cycles ([3]) are both incompatible with partial order reduction, such as proposed in [5, 4]. We have discussed a modification of one of the two algorithms that secures compatibility with reduction methods, and that suffices to solve both search problems efficiently and uniformly.

Acknowledgement: The authors are most grateful to Michael Ferguson, whose keen

```
proc dfs(s)
    if error(s) then report error fi
    add {s,0} to Statespace
    for each successor t of s do
       if {t,0} not in Statespace then dfs(t) fi
    od
    ndfs(s) /* different */
end
proc ndfs(s) /* the nested search */
    if s is Progress State then return fi /* new */
    add {s,1} to Statespace
    add s to Stack /* new */
    for each successor t of s do
       if {t,1} not in Statespace then ndfs(t) fi
       else if t is in Stack then report cycle fi /* different */
    od
    delete s from Stack /* new */
end
```

Figure 5: Nested Depth First Search for Non-Progress Cycles

6 Absence of Starvation

In [3] a different version of a nested depth-first search is described to solve a slightly different type of problem. The problem here is the detection of cycles in the reachability graph that do *not* contain any user-defined *progress* states. Any infinite execution that contains only finitely many traversals of progress states corresponds to starvation.

The algorithm from [3] is shown in Figure 5 (shown here in the same format as Figures 1 and 2). The differences with Figure 2 are as follows. From every reachable state we start the nested search, but in the second search no traversals through progress states are allowed. Whenever a cycle is closed on the stack from this second depth-first search, an error can be reported. In this variant of the search, therefore, also the non-reduced variant relies on information from the search stack. (Note that an explicit search stack must be maintained here only for the *second* search, while in Figure 4 the explicit stack must be maintained only for the *first* search.)

A direct combination of this algorithm with the reduction strategy from [5] fails for the same reasons as before, but this time the correction also fails. Note that when the second search intersects the search stack from the first depth-first search, there does not necessarily exist a valid path back to the root of the second depth-first search (the seed state). (The path that exists is invalid if it contains progress states.)

```
proc dfs(s)
    if error(s) then report error fi
    add {s,0} to Statespace
    add s to Stack
    for each (selected) successor t of s do
      if {t,0} not in Statespace then dfs(t) fi
    od
    if accepting(s) then ndfs(s) fi
    delete s from Stack
end
proc ndfs(s) /* the nested search */
    add {s,1} to Statespace
    for each (selected) successor t of s do
      if {t,1} not in Statespace then ndfs(t) fi
      else if t in Stack then report cycle fi
    od
end
```

Figure 4: Optimized Nested Depth First Search, Compatible with Reduction

- Add an integer that holds a "selection number." The partial order reduction algorithm can make several attempts to select a safe (reduced) subset of successor states. In SPIN, these selections correspond to process numbers [4]. When all attempts to chose a safe selection fail, a complete expansion of all successors is done. The first depth-first search would keep the selection number for the use of the second depth-first search. The second depth-first search would generate the sets of successors (*ample sets*, in the terminology of [5]) according to the same order, and use the selection number that was held by the first depth-first search, ignoring its own cycle closing. A selection number of zero would mean selecting all successors from the current state.

- A more modest solution uses only one bit as a selection number. In this case, if a selection caused closing a cycle in the first depth-first search, no alternative subset is sought, and the entire set of successors is generated from the current node. The selection bit communicates to the second depth-first search whether to use its first subset or to do a full successor expansion.

The addition of the selection bit is a small modification of the reduction algorithm discussed in [4] that we will not elaborate further here.

5 The Correction

To correct the algorithm it will suffice if we can guarantee that the second depth-first search always explores the same states that are found in the first depth-first search. We will present two changes to accomplish this.

5.1 Intersection with Search Stack from First Search

The problem in Figure 3 occurs when the second depth-first search reaches a state that exists also on the first search stack (state s_1). Continuing the second search from this state, the search can now reach states with a different (longer) search stack in the second depth-first search than in the first. This means that the third condition from the partial order reduction algorithm is applied to different states, and the second search might not follow the first search. Where the sets of successor states that is selected in the two phases of the search can differ.

The solution to this problem is remarkably simple: when the second depth-first search reaches a state that exists also on the stack from the first search, the search can be *terminated*. The reason is that reaching such a state from the seed state implies immediately that a path exists that leads back to the seed state: it is the path of states on the first depth-first search stack that starts at the point of intersection. This property is independent of the use of the partial order reduction. Stopping the search when reaching a state on the first search stack shortens the search, it improves the performance of the algorithm, and it allows for shorter counter-examples to an invalid correctness claim to be generated.

The change in the algorithm of Figure 2 from [1] is shown in Figure 4. In the new version of this algorithm, there is no longer a need to store the seed state, but we do need information about the presence of states on the search stack.

5.2 Preserving Information Between Searches

The first correction of the nested depth-first search algorithm improves its performance and it eliminates the error that occurred in Figure 3. But, it does not completely solve the problem.

A failure can still happen. First note that the first depth-first search can backtrack from a graph component without performing a second search. (The second search is only initiated from accepting states.) While searching another component of the graph, that does contain accepting states, there can be a transition that leads back to the earlier states, that do not as yet appear in the second state space. This return transition would not be followed in the first search, since it leads only to previously visited states. During a second search, however, the transition must be followed, A similar scenario now exists, in which the states are visited with a different search stack in each of the two searches, and the reachability properties are not preserved.

To eliminate also this problem, one must preserve information about the selection of successors between the two searches. There are several possibilities:

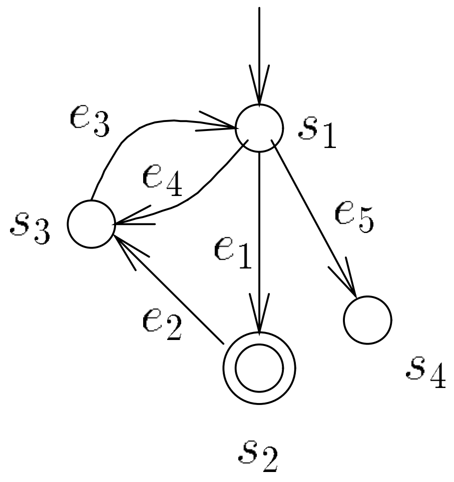

Figure 3: Failure of the Reduced Nested Depth-First Search

that is based on this condition can depend on the precise search order and the contents of the search stack at each point during the search.

A direct combination of the partial order reduction technique from [5, 4] with the nested depth-first search method from [1] therefore results in an incorrect algorithm. Since the contents of the depth-first search stacks in the first and in the second search in the nested depth-first search algorithm from Figure 2 differ, the state space that is constructed during the first and the second part of the search need no longer be equal during a reduced search. This means that the second search may be incapable of proving that an accepting state is reachable from itself, even if it is part of a strongly connected component in the reachability graph that is constructed during the first search.

The example in Figure 3 illustrates the problem. Consider the case where the first search reaches state s_1. Its successors are s_2 and s_3. In the full state space, there is another successor s_4, but it is not selected. First edge e_1 is taken, leading to accepting state s_2. Now, the first search continues to search all the nodes accessible from s_2. When backtracking to s_2, the second search starts, with s_2 as its seed state, looking for a cycle. Such a cycle exists by taking the edges e_2, e_3 and then e_1 again. However, once reaching s_1 during the second search, a cycle consisting of e_3 and e_4 is found. Notice that this cycle was not found in the first search, as s_3 was not on the search stack of the first search while searching for successors of s_1. Closing this cycle with the edge e_4 in the second search makes the second search select an alternative subset of successors, e.g., s_4, via the edge e_5. But now, it is possible that the cycle through seed state s_2 remains undetected.

```
proc dfs(s)
    if error(s) then report error fi
    add {s,0} to Statespace
    for each successor t of s do
      if {t,0} not in Statespace then dfs(t) fi
    od
    if accepting(s) then seed:=s; ndfs(s) fi
end
proc ndfs(s) /* the nested search */
    add {s,1} to Statespace
    for each successor t of s do
      if {t,1} not in Statespace then ndfs(t) fi
      else if t==seed then report cycle fi
    od
end
```

Figure 2: Nested Depth-First Search

constructed by the algorithm from Figure 1 suffices to separate the two searches. That is: the size of Statespace need not double as initially suggested in [1], but can remain virtually unchanged. The search time, however, can double when all states are reachable in both the first and the second search, and no cycles through accepting states exist.

Notice that also in the modified algorithm, there is still no need to store the edges of the graph, nor to access the depth-first search stack.

4 Reduced Search

The statespace that is constructed with the depth-first search procedure can be reduced substantially if we limit the number of successor states that is explored from each reachable state.

In [5] a reduction of this type is discussed that preserves the Büchi acceptance properties from the full statespace also in the reduced statespace, provided that three conditions are satisfied. Two of the three conditions deal with dependency between execution steps, and the visibility of individual execution steps. They do not alter the properties of either the basic or the nested the depth-first search.

A third condition, however, introduces a dependency on the information that is stored in the depth-first search stack. The condition states that a reduction of the set of successor states from state s is invalid if at least one of those successor states appears on the depth-first search stack. This means that reachability properties during a *reduced* search

```
proc dfs(s)
    if error(s) then report error fi
    add s to Statespace
    for each successor t of s do
        if t not in Statespace then dfs(t) fi
    od
end
```

Figure 1: Basic Depth-First Search Algorithm, Used For Reachability Analysis

2 Basic Depth-First Search

A basic depth-first search algorithm generates and examines every global state that is reachable from a given initial state ι, as illustrated in Figure 1.

After a first check for the validity of the state and its properties, the state descriptor is entered into a global Statespace, usually with the help of standard hash-table lookup procedures. A recursive call to the search procedure is then made for each state that is reachable from this state in one atomic execution step, i.e., for each possible successor in the reachability graph that is not already represented in Statespace.

Note that Statespace needs to represent only the nodes of the reachability graph; the representation of the edges between the nodes is not needed. Note also that no information is needed about the presence of any state on the depth-first search stack.

3 Nested Depth-First Search

The procedure from Figure 1 cannot be used directly to solve a model checking problem, but it can readily be adapted. In model checking, the reachability graph, partially represented in the Statespace structure, is used to define a Büchi automaton. The acceptance conditions in this automaton are usually derived from a special property automaton that is added to the global system, as described in [6, 3].

The model checking problem in this context reduces to the problem of detecting the existence in the graph of accepting states that are part of a strongly connected component and therefore reachable from themselves.

The procedure in [1] applies a nested depth-first procedure to solve this problem, as illustrated in Figure 2. When the normal search retracts to an accepting state, a second search is started, to search for a cycle through this state. If the second search fails to find a cycle, the first search resumes from the point where it was interrupted. It is shown in [2] that adding two bits to every global state that is stored in the Statespace that is

On Nested Depth First Search

Gerard Holzmann, Doron Peled, Mihalis Yannakakis
Bell Laboratories
700 Mountain Avenue
Murray Hill, NJ 07974
{gerard, doron, mihalis}@research.bell-labs.com

Extended Abstract

Abstract

We show in this paper that the algorithm for solving the model checking problem with a nested depth-first search proposed in [1] can interfere with algorithms that support partial order reduction. We introduce a revised version of the algorithm that guarantees compatibility. The change also improves the performance of the nested depth-first search algorithm when partial order reduction is not used.

1 Introduction

The model checking problem for concurrent systems can be solved elegantly by modeling the system and its correctness requirements as automata on infinite words (i.e., Büchi Automata), and by verifying that the language of the product of system and requirement automata is empty. The emptiness problem itself would normally require the computation of the strongly connected components of the product system. This computation can be avoided with a nested depth-first search procedure, as discussed in [1, 2, 3]. The memory requirements of a reachability analysis, based on either depth-first or breadth-first search order, can be reduced further with the help of property preserving reduction strategies. One such method was described in [5, 4]. We recently discovered that the two methods are not completely compatible, which means that a direct combination will result in an incomplete model checking procedure.

Section 2 first gives a synopsis of the basic and nested depth-first search methods. Section 3 discusses reduction, and illustrates the undesirable side-effects of a direct combination of the algorithms. Section 4 describes a modification of the nested depth-first search algorithm that avoids the problem. In Section 5, finally, we discuss a variation of the algorithm from [3] that can be used for proving absence of starvation, but that does not appear to be compatible with the reduction strategy. An alternative technique, that is compatible, is also discussed.