Improving Non-progress Cycle Checks

David Faragó* and Peter H. Schmitt

Universität Karlsruhe (TH), Institut für Theoretische Informatik
Logik und Formale Methoden
{farago,pschmitt}@ira.uka.de

Abstract. This paper introduces a new model checking algorithm that searches for non-progress cycles, used mainly to check for livelocks. The algorithm performs an incremental depth-first search, i.e., it searches through the graph incrementally deeper. It simultaneously constructs the state space and searches for non-progress cycles. The algorithm is more efficient than the method the model checker SPIN currently uses, and finds shortest (w.r.t. progress) counterexamples. Its only downside is the need for a subsequent reachability depth-first search (which is not the bottleneck) for constructing a full counterexample. The new algorithm is better combinable with partial order reduction than SPIN's method.

Key words: Model Checking, SPIN, Non-progress cycles, livelocks, depth-first search, partial order reduction.

1 Introduction

In Section 1.1, we describe what non-progress cycles (*NPCs*) are and how SPIN currently searches for them. Section 1.2 presents SPIN's method in more detail and reveals its redundant operation. Hence we apply a new idea (see Section 2.1) to design two new algorithms (the incremental DFS and DFS_{FIFO}, see Section 2.2), which are the main contribution of this paper. After proving the correctness of DFS_{FIFO} in Section 2.3, we show in Section 2.4 that it has several advantages over SPIN's method. The section ends by depicting the high relevance of partial order reduction. After describing how this reduction works (see Section 3.1), we show that its usage by DFS_{FIFO} is correct (see Section 3.2) and yields many further advantages (see Section 3.3). The paper closes with a conclusion and future work.

1.1 Non-progress Cycle Checks by SPIN

NPC checks are mainly used to detect livelocks in the system being modeled, i.e., execution cycles that never make effective progress. NPC checks are often performed in formal verifications of protocols, where livelocks frequently occur. Using SPIN, livelocks were found, for instance, in the i-protocol from UUCP (see

* This research received financial support by the Concept for the Future of KIT within the framework of the German Excellence Initiative from DFG.

[3]) and GIOP from CORBA (see [10]), whereas DHCP was proved to be free of livelocks (see [9]). To be able to check for NPCs, desired activities of the system are marked in PROMELA by labeling the corresponding location in the process specification with a progress label: "statement$_i$; progress: statement$_j$;". This sets the local state between statement$_i$ and statement$_j$ to a *local progress state* (cf. Figure 7). A *(global) progress state* is a global system state in which at least one of the processes is in a local progress state. SPIN marks global progress states by setting the global variable *np_* to false. Section 2.4 presents progress transitions as alternative for modeling progress. If no cycle without any progress-label exists, then the system definitely makes progress eventually (it never gets stuck in a livelock).

A *non-progress cycle check* detects (and returns a path to) a reachable *non-progress cycle*, i.e., a reachable cycle with no progress states, iff there exists one in the state transition system (S, T) (with S being the set of states and $T \subseteq S \times S$ the set of transitions).

SPIN puts the check into effect with the *Büchi automaton* for the LTL formula $\Diamond \Box\, np\_$, which translates into the *never claim* of Listing 1 (cf. [6]).

```
never { /* <>[] np_ */
  do /*nondeterministically delay or swap to NPC search mode*/
  :: np_ -> break
  :: true /*nondeterministic delay mode*/
  od;
accept: /*NPC search mode*/
  do
  :: np_
  od
}
```

Listing 1. Never claim for NPC checks

The LTL formula is verified with SPIN's standard *acceptance cycle check*, the *nested depth-first search* (*NDFS*) algorithm (see [8,6]): Before the basic *depth-first search* (*DFS*) backtracks from an accepting state s and removes it from the stack, a second, nested DFS is started to check whether s can reach itself, thus resulting in an acceptance cycle. Pseudo-code for the nested DFS is given in Listing 2.

1.2 Motivation for Improving Non-progress Cycle Checks

The following walkthrough depicts a detailed NPC check in SPIN (cf. Figure 1):

1. When traversal starts at *init*, the never claim immediately swaps to its NPC search mode because the never claim process firstly chooses *np_* −> break in the first **do**-loop (if the order in this **do**-loop were swapped, the NPC

```
proc DFS(state s)
  if error(s) then report error fi;
  add {s,0} to hash table;
  push s onto stack;
  for each successor t of s do
    if {t,0} ∉ hash table then DFS(t) fi
  od;
  if accepting(s) then NDFS(s) fi;
  pop s from stack;
end

proc NDFS(state s) /* the nested search */
  add {s,1} to hash table;
  for each successor t of s do
    if {t,1} ∉ hash table
    then NDFS(t)
    else if t ∈ stack then report cycle fi
    fi
  od;
end
```

Listing 2. Nested DFS

check would descend the graph as deeply as possible in the nondeterministic delay mode). Hence a DFS is performed in which all states are marked as acceptance states by the never claim and progress states are omitted, i.e., truncated (see Listing 1).

2. Just before backtracking from each state being traversed in the NPC search mode DFS, the NDFS (i.e., the nested search) starts an acceptance cycle search (since all traversed states were marked as acceptance states). For these acceptance cycle searches, the non-progress states are traversed *again*.
3. If an acceptance cycle is found, it is also an NPC since only non-progress states are traversed. If no acceptance cycle is found, the NDFS backtracks from the state s where the NDFS was initiated, but *immediately starts a new NDFS* before the NPC search mode DFS backtracks from the predecessor of s. Fortunately, states that have already been visited by an NDFS are not revisited. But the NDFS is repeatedly started many times and at least one transition has to be considered each time. Eventually, when the NDFS has been performed for all states of the NPC search mode DFS, the NPC search mode DFS backtracks to *init*.
4. No the nondeterministic delay mode DFS constructs the state space *once more*. During this, after each forward step, *all previous procedures are repeated*. Since most of the time the states have already been visited, these procedures are immediately aborted. During this nondeterministic delay mode DFS, also progress states are traversed.

On the whole, the original state space (i.e., without the never claim) is traversed three times: in the NPC search mode DFS, in the NDFS and in the nondeterministic delay mode DFS. The state space construction for reaching an NPC and the NPC search are performed in separate steps.

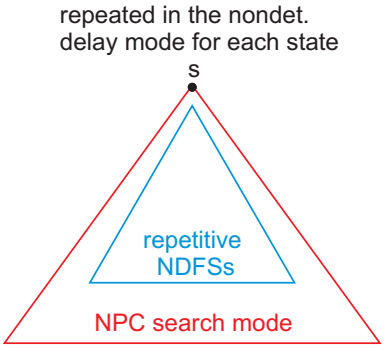

Fig. 1. Walkthrough of SPIN's NPC check

2 Better NPC Checks

In this section, we firstly introduce our new approach and then our new algorithms: the incremental DFS and its improvement $\mathrm{DFS}_{\mathrm{FIFO}}$. Thereafter, the correctness of $\mathrm{DFS}_{\mathrm{FIFO}}$ is proved. Finally, it is compared to SPIN's method of NPC checks.

2.1 Approach

The detailed walkthrough in Section 1.2 has shown that SPIN's NPC check unnecessarily often initializes procedures, touches transitions, and traverses the state space. The cause for this inefficiency is the general approach: acceptance cycle checks in combination with never claims are very powerful and cover more eventualities and options than necessary for NPC checks. So we are looking for a more specific algorithm for NPC checks that performs less redundantly and combines the construction and the NPC search phase.

But with only one traversal of the state space, we have to cope with the following problem: Simply checking for each cycle found in a basic DFS whether it makes progress is an incomplete NPC search since the DFS aborts traversal in states which have already been visited. Hence not all cycles are traversed. Figure 2 shows an example of an NPC that is not traversed and therefore not found by the DFS: From s_1, the DFS first traverses *path 1*, which contains a progress

state s_2. After backtracking from *path 1* to s_1, the DFS traverses *path 2*, but aborts it at s_3 before closing the (red, thick) NPC. Hence if an NPC has states that have already been visited, the cycle will not be found by the basic DFS.

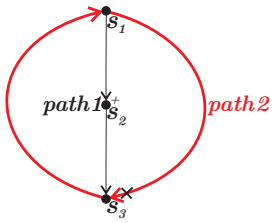

Fig. 2. Not traversed NPC

The idea for our alternative NPC checks is to guarantee that:

(*p-stop*) After reaching an NPC for the first time, traversal of progress states is postponed as long as possible.

This constraint enforces that NPCs are traversed before states from the NPC are visited through some progress cycle and thus break the NPC traversal. The following section introduces two new algorithms and checks (p-stop) for them.

2.2 The Incremental Depth-First Search Algorithms

Incremental Depth-First Search. This new algorithm searches for NPCs using a depth-first iterative deepening search with incrementally larger thresholds for the number of progress states that may be traversed. For this, the *incremental DFS* algorithm, described in Listing 3 and 4, repeatedly builds subgraphs G_L where paths (starting from *init*) with maximally L progress states are traversed, for $L = 0, 1 \ldots$, with a basic DFS algorithm. It terminates either with an *error path* (a counterexample to the absence of NPCs) when an NPC is found or without an error when L becomes big enough for the incremental DFS to build the complete graph G, i.e., $G_L = G$.

So in each state s we might prune some of the outgoing transitions by omitting those which exceed the current progress limit L, and only consider the remaining transitions.

We can implement $\text{DFS}_{\text{starting\_over}}$ by using a *progress_counter* for the number of progress states on the current path. The progress_counter is saved for every state on the stack, but is ignored when states are being compared. This causes an insignificant increase in memory (maximally $\log(L_{max}) \times \text{depth}(G)$ bits). With this concept, we can firstly update the progress_counter when backtracking, secondly abort traversal when the progress_counter exceeds its current limit L, and thirdly

```
proc DFS_starting_over(state s)
  L:=0;
  repeat
    DFS_pruned:=false;
    DFS_prune,NPC(s);
    L++;
  until (!DFS_pruned);
end;

proc main()
  DFS_starting_over(init);
  printf("LTS does not contain NPCs");
end;
```

Listing 3. Incremental depth-first search

quickly check for progress whenever a cycle is found. To complete this implementation, we still have to determine the unsettled functions of $\mathrm{DFS}_{\mathrm{prune,NPC}}$, underlined in listing 4:

- $\underline{\text{pruned}}(s,t)$ returns true iff progress_counter $= L$ and t is a progress state.
- $\underline{\text{pruning\_action}}(t)$ sets DFS_pruned to true.
- $\underline{\text{np\_cycle}}(t)$ returns true iff progress_counter = (counter on stack for t).
- The $\underline{\text{error\_message}}$ can print out the stack, which corresponds to the path from *init* to the NPC, inclusively.

Unfortunately, this incremental DFS has several deficiencies:

- The upper part of the graph (of the graph's computation tree) is traversed repeatedly. But since we usually have several transitions leaving a state and relatively few progress states, this makes the incremental DFS require maximally twice the time of one complete basic DFS (i.e., of a safety check).
- Depending on the traversal order of the DFS, the progress_counter limit might have to become unnecessarily large until an NPC is found, cf. Figure 3.
- As main disadvantage, the approach of the incremental DFS is not sufficient to fulfill the condition (p-stop): It can happen that a state s_0 on an NPC is reached the first time for progress_counter limit L_0 (via *path 3*), but with progress_counter$(s_0) < L_0$. For this to be the case, *path 3* was aborted for $L < L_0$. Hence for $L < L_0$, a state s_1 on *path 3* was already visited from another path (*path 2*) with more progress, see Figure 3. For L_0, s_0 was reached via *path 3*, and thus *path 2* was pruned. Therefore a state s_2 on *path 2* has already been visited via another path (*path 1*) for L_0, but not for $L < L_0$. This situation is depicted in Figure 3, with the traversal order equal to the path number, $L_0 = 3$, progress_counter$(s_0) = 2$, and $+$ marking progress.

Hence we modify the incremental DFS algorithm in the following section.

```
proc DFS_prune,NPC(state s)
  add s to hash table;
  push s onto stack;
  for each successor t of s do
    if (t ∉ hash table)
    then if (!pruned(s,t))
         then DFS_prune,NPC(t)
         else pruning_action(t)
         fi
    else if (t ∈ stack && np_cycle(t))
         then halt with error_message
         fi
    fi
  od;
  pop s from stack;
end;
```

Listing 4. Generic DFS with pruning and NPC check

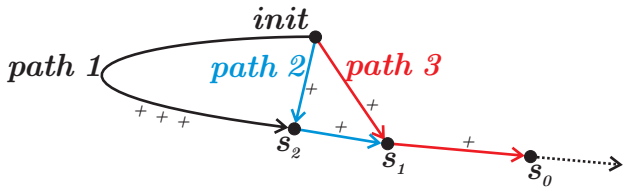

Fig. 3. (p-stop) is not met for the incremental DFS

Incremental Depth-First Search with FIFO. Instead of repeatedly increasing the progress_counter limit L and re-traversing the upper part of the graph's computation tree, we save the pruned progress states to jump back to them later on to continue traversal. Roughly speaking, we perform a breadth-first search with respect to the progress states, and in-between progress states we perform DFSs (cf. Listing 5).

To reuse the subgraph already built, we have to save some extra information to know which transitions have been pruned. One way to track these pruned transitions is by using a FIFO (or by allowing to push elements *under* the stack, not only on top). Hence we name the algorithm incremental DFS with FIFO (DFS_{FIFO}). Listing 5 shows that we do not repeatedly construct the graph from scratch, but rather use the graph already built, gradually pick the progress states out of the FIFO, and expand the graph further by continuing the basic DFS. When a new progress state is reached, traversal is postponed by putting the state into the FIFO. When the basic DFS is finished and the FIFO is empty, the complete graph G is built.

```
proc DFS_FIFO(state s)
  put s in FIFO;
  repeat
    pick first s out of FIFO;
    DFS_prune,NPC(s)
  until (FIFO is empty);
end;

proc main()
  DFS_FIFO(init);
  printf("LTS does not contain NPCs");
end;
```

Listing 5. Incremental depth-first search with a FIFO

The unsettled functions of $\mathrm{DFS}_{\mathrm{prune,NPC}}$ are defined for $\mathrm{DFS}_{\mathrm{FIFO}}$ as follows:

- pruned(s,t) returns true iff t is a progress state.
- pruning_action(t) puts t into the $FIFO$.
- np_cycle(t) returns (t != first element of stack), since the first element is a progress state. (Using progress transitions (see Section 2.4), this exception becomes unnecessary and the constant **true** is returned.)
- The error_message can print out the stack, which now corresponds to the path of the NPC found, but no longer contains the path from *init* to the cycle.

Note. This algorithm does not know which G_L is currently constructed. If we want to clearly separate the different runs, as before, we can use two FIFOs, one for reading and one for writing. When the FIFO that is read from is empty, the current run is finished and we swap the FIFOs for the next run.

With this technique, the deficiencies from the original incremental DFS are avoided: (p-stop) is fulfilled since progress state traversal is postponed as long as possible, the progress counter limit L does not become unnecessarily large, and we avoid the redundancy of the original incremental DFS by reusing the part of the graph already built previously. The consequent postponing guarantees a constraint stronger than (p-stop): Each state is visited through a path with the fewest possible progress states. So now G_L is the *maximal* subgraph of G such that all paths in G_L without cycles have at most L progress states.

On the whole, $\mathrm{DFS}_{\mathrm{FIFO}}$ does not require more memory by using a FIFO compared to a basic DFS because progress states are stored only temporarily in the FIFO until they are stored in the hash table (cf. Listing 4). The time complexity is also about the same as for the basic DFS.

$\mathrm{DFS}_{\mathrm{FIFO}}$ erases a large part of the stack: everything behind progress states, i.e. all of the stack between *init* and the last progress state, is lost. But for detecting NPCs, this is a feature and not a bug: Exactly the NPCs are detected. The cycles that go back to states from previous runs are progress cycles and stay undetected. Thus we no longer need to save a progress counter on the truncated

stack, saving even more memory. A further benefit will arise in combination with partial order reduction (see Section 3).

If an NPC is detected, the stack from the current run supplies the NPC, but an additional basic DFS for reaching the NPC is required to obtain a complete error path as counterexample. The shortest (w.r.t. progress) counterexample can be found quickly, for instance with the following method, which generally requires only little additional memory: Instead of storing only the last progress states in the FIFO, all progress states on the postponed paths are saved, e.g., in form of a tree. The shortest counterexample can then easily be reconstructed using the progress states on the error path as guidance for the additional basic DFS.

2.3 Correctness of $\mathbf{DFS_{FIFO}}$

Constructively proving that $\mathrm{DFS_{FIFO}}$ finds a certain NPC is difficult: We would require to consider various complex situations and the technical details of the algorithm, e.g., the order in which the transitions are traversed (cf. Figure 6). Hence we prefer a pure existence proof.

Theorem 1. *DFS_{FIFO} finds an NPC if one exists and otherwise outputs that no NPC exists. An NPC is found at the smallest depth w.r.t. progress, i.e., after the smallest number (L_0) of progress states that have to be traversed.*

Proof. $\mathrm{DFS_{FIFO}}$ only postpones transitions, but does not generate new ones. It checks for NPCs by searching through the stack (except the first state), i.e., it only considers non-progress states. Thus it is sound, i.e., it does not output false negatives.

To prove completeness, i.e., that NPCs are found if they exist, let there be an NPC. As long as $\mathrm{DFS_{FIFO}}$ constructs G_L for $L < L_0$, all paths leading to an NPC are pruned. Let $L = L_0$, s be the first state reached in the DFS which is on an NPC, t_{begin} be the time the DFS reaches s (the first time), and $\pi^1 = \langle\, s_1^1{=}s, s_2^1, \ldots, s_{n_1}^1{=}s \,\rangle$ be an NPC containing s. Because of (p-stop), the DFS stops traversing progress states while all non-progress states reachable from s are being traversed.

We assume no NPC is found in G_{L_0}. Hence the traversal of π^1 must be aborted because a state $s_{h_1}^1 \neq s$ for $h_1 \in \{2, \ldots, n_1 - 1\}$ is revisited (i.e., visited when already in the hash table) before π^1 is closed, i.e., before s could be reached the second time. Let t_{middle_1} be the time when $s_{h_1}^1$ is visited the first time and t_{end} be the time when $s_{h_1}^1$ is revisited and π^1 is aborted. $s_{h_1}^1$ cannot be twice on the current path (once at the end and once earlier) at time t_{end}: the first occurrence cannot be above s (i.e., closer to *init*) because s is the first visited state of π^1, and not below s, since then an NPC $\langle s_{h_1}^1, \ldots, s_{h_1}^1 \rangle$ would be found, see Figure 4. So our algorithm first visits s in t_{begin}, then visits $s_{h_1}^1$ at t_{middle_1}, then backtracks from $s_{h_1}^1$ and finally revisits $s_{h_1}^1$ at t_{end} while traversing π^1.

Informally, since our algorithm backtracks from $s_{h_1}^1$ without having found an NPC, the traversal of the path from $s_{h_1}^1$ to s was aborted because some state on

it was revisited, i.e., the state has already been visited before, but after t_{begin}. With this argument, we come successively closer to completing some NPC, which is a contradiction.

Formally: Let $\pi^2 = \langle s_1^2{=}s, s_2^2, \ldots, s_{n_2}^2{=}s \rangle$ be the path from s at time t_{begin} to $s_{h_1}^1$ at time t_{middle_1} concatenated with $\langle s_{h_1+1}^1, s_{h_1+2}^1, \ldots, s_{n_1}^1{=}s \rangle$, i.e., π^2 is also an NPC containing s, see Figure 5. Therefore we can apply the argumentation from above to π^2 instead of π^1 to obtain a state s_k^2 ($k \in \{1, \ldots, n_2\}$) on π^2 that is revisited before π^2 is closed. Let t_{middle_2} be the time when s_k^2 is visited the first time. Since on π^2 the state $s_{h_1}^1$ is visited the first time (at t_{middle_1}), the DFS also reaches (maybe a revisit) $s_{h_1+1}^1$ on π^2 (at some time after t_{middle_1}). So $s_k^2 = s_{h_2}^1$ for $h_2 \in \{h_1+1, \ldots, n_1-1\}$. Let $\pi^3 = \langle s_1^3{=}s, s_2^3, \ldots, s_{n_3}^3{=}s \rangle$ be the NPC from s at time t_{begin} to $s_{h_2}^1$ at time t_{middle_2} concatenated with $\langle s_{h_2+1}^1, s_{h_2+2}^1, \ldots, s_{n_1}^1{=}s \rangle$. Applying this argumentation iteratively, we get a strictly monotonically increasing sequence $(h_i)_{i \in \mathbb{N}}$ with all $h_i \in \{2, \ldots, n_1 - 1\}$. Because of this contradiction, the assumption that no NPC is found in G_{L_0} is wrong.

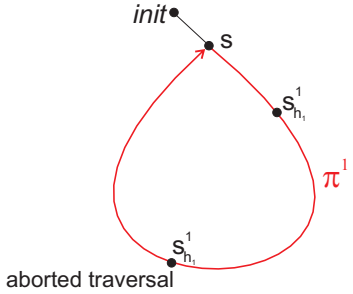

Fig. 4. $s_{h_1}^1$ cannot be twice on the current path at time t_{end}

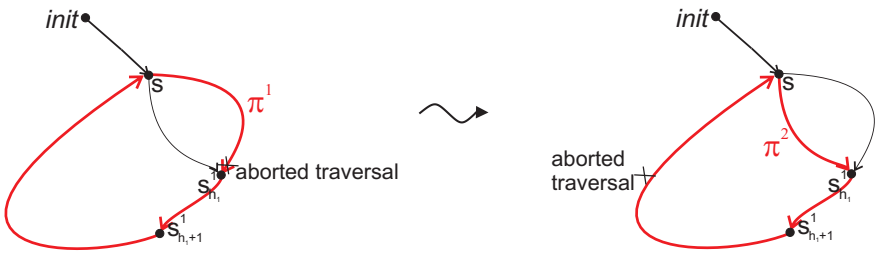

Fig. 5. Constructing π^2 from π^1

If all cycles in the LTS make progress, L will eventually be big enough to contain the complete graph. After traversing it the algorithm terminates with the correct output that no NPC exists. Thus our algorithm is correct.

Note. The proof shows that DFS_{FIFO} finds an NPC before backtracking from s. But the NPC does not have to contain s: Figure 6 shows an example if t_1 is traversed ahead of t_2. Since our pure existence proof assumed that no NPCs are found, it also covers this case (cf. Figure 4).

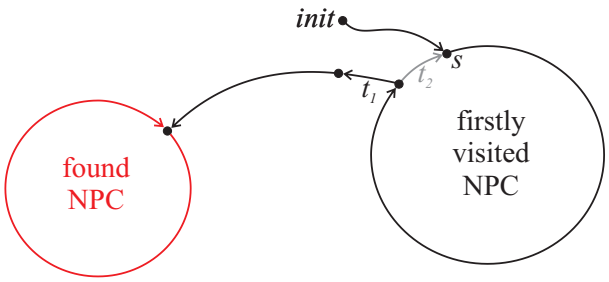

Fig. 6. The found NPC does not contain s

2.4 Comparison

We firstly compare SPIN's NPC checks with DFS_{FIFO} by relating their performances to that of the basic DFS (which corresponds to a safety check): The runtime for a basic DFS, denoted t_{safety}, is linear in the number of reachable states and transitions, the memory requirement s_{safety} is linear in the number of reachable states. For SPIN's NPC checks, the memory required in the worst case is about $2 \times s_{\text{safety}}$ because of the never claim. The runtime is about $3 \times t_{\text{safety}}$ because of the nested search and the doubled state space. For DFS_{FIFO}, both time and memory requirements are about the same as for the basic DFS. To construct a full counterexample, maximally t_{safety} is required, but usually far less.

But this asymptotic analysis only gives rough complexities. For a more precise comparison, we look at the steps in detail and see that the inefficiencies of the NDFS algorithm are eliminated for DFS_{FIFO}: The redundancy is completely avoided as all states are traversed only once by a simultaneous construction and NPC search.

Furthermore, only paths with minimal progress are traversed. Since many livelocks in practice occur after very little progress – e.g., for the i-protocol (cf. [3]) after 2 sends and 1 acknowledge – DFS_{FIFO} comprises an efficient search heuristic. Additionally, shortest (w.r.t. progress) counterexamples are easier to understand and often reveal more relevant errors.

Finally, we can also model progress in a better way using *progress transitions* instead of progress states. SPIN's NPC check needs to mark states as having progress because never claims are used: The never claim process is executed in lockstep with the remaining automaton and thus only sees the states of the remaining automaton, not its transitions. Since our $\mathrm{DFS}_{\mathrm{FIFO}}$ does not require never claims, we can mark transitions (e.g., those switching $np\_$ from false to true in the original semantics) as having progress. The most fundamental implementation of progress transitions is to change the semantics of PROMELA so that a progress label marks the following statement as a progress transition. If we do not want to change the PROMELA semantics, we can use the construct "statement$_i$; atomic {skip; progress: statement$_j$}" instead of "statement$_i$; progress: statement$_j$;". Figure 7 shows the difference in the automata: The progress moves from state s to the following composite transition. "atomic{...}" guarantees that the progress state is left immediately after it was entered. Unfortunately, SPIN does not interleave atomic sequences with the never claim process, so this technique cannot be used for SPIN's NPC checks. Trying nevertheless, SPIN sometimes claims to find an NPC, but returns a trace which has a progress cycle; At other times, SPIN gives the warning that a "`progress label inside atomic - is invisible`". SPIN's inconsequent warning suggests that it does not always detect progress labels inside atomic.

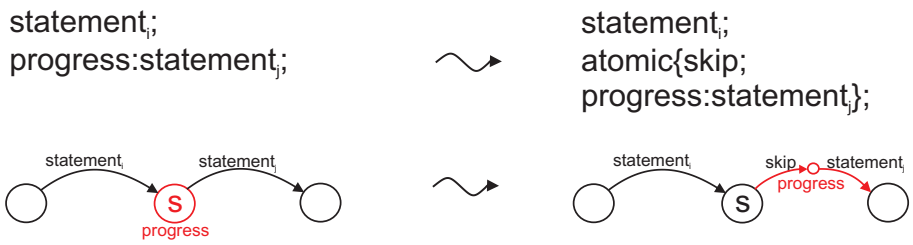

Fig. 7. Progress transition with atomic

Using progress transitions, we can model more faithfully since in reality actions, not states, make progress. For example in Figure 8, if the action corresponding to the transition from a state s_2 to a state s_1 causes progress, PROMELA models s_2 as progress state. So the cycle between (s_2, t_1) and (s_2, t_2) in the global automaton is considered as progress cycle, although the system does not perform any progress within the cycle. The other case of a path with several different local progress states visited simultaneously or directly after one another cannot be distinguished from one persistent local progress state as in Figure 8. Using progress transitions, all these cases can be differentiated and are simpler: The number of progresses on a path π is simply its number of progress

transitions, denoted $|\pi|_p$. The biggest advantages of using progress transitions emerge in combination with partial order reduction (see Section 3).

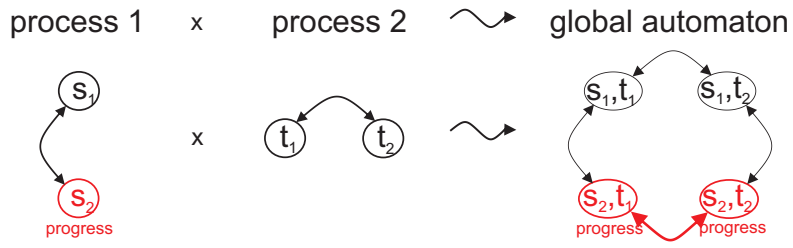

Fig. 8. Faked progress

The performance comparison from this section has to be considered with caution, though, as the effectiveness of a verification usually stands and falls with the strength of the additional optimization techniques involved, especially partial order reduction (cf. Section 3). The reduction strength can significantly decrease when changing from safety to liveness checks because the traversal algorithm changes and the visibility constraint C2 (cf. Section 3) becomes stricter. For instance, in a case study that verified leader election protocols (cf. [4]), the safety checks with partial order reduction were performed in quadratic time and memory (i.e., easily up to SPIN's default limit of 255 processes), whereas the NPC checks could only be performed up to the problem size 6 (see Table 1). So a very critical aspect of NPC checks is how strongly partial order reduction can reduce the state space.

Table 1. Big difference between safety and NPC checks

Problem	safety checks			NPC checks via NDFS		
Size	time	depth	states	time	depth	states
3	5”	33	66	5”	387	1400
4	5”	40	103	5”	2185	18716
5	5”	47	148	6”	30615	276779
6	5”	54	201	70”	335635	4.3e+06
7	5”	61	262	memory overflow (> 1GB)		
254	100”	1790	260353	memory overflow (> 1GB)		

In the next section, we show that DFS_{FIFO} is compatible with partial order reduction and that the elimination of redundancy in DFS_{FIFO} – as well as its further advantages (see Section 3.3) – enhance the strength of partial order reduction.

3 Compatibility with Partial Order Reduction

SPIN's various reduction methods contribute strongly to its power and success. Many of them are on a technical level and easily combinable with our NPC checks, for instance Bitstate Hashing, Hash-compact and compression.

One of the most powerful reduction methods is *partial order reduction* (*POR*). In this section, we firstly introduce SPIN's POR, which uses the technique of ample sets (see [1,2], for technical details [7]). Thereafter, we prove that $\mathrm{DFS}_{\mathrm{FIFO}}$ can be correctly combined with POR. Finally, we again compare SPIN's NPC checks with $\mathrm{DFS}_{\mathrm{FIFO}}$, this time also considering POR.

3.1 Introduction

One of the main reasons for state space explosion is the interleaving technique of model checking to cover all possible executions of the asynchronous product of the system's component automata. These combined executions usually cause an exponential blowup of the number of transitions and intermediate states. But often statements of concurrent processes are independent:

$\alpha, \beta \in \mathfrak{S}$ are *independent*	iff	$\forall s \in S : \alpha, \beta \in \mathit{enabled}(s) \implies$ $\alpha \in \mathit{enabled}(\beta(s))$ and $\beta \in \mathit{enabled}(\alpha(s))$ (*enabledness*) and $\alpha(\beta(s)) = \beta(\alpha(s))$ (*commutativity*)
$\alpha, \beta \in \mathfrak{S}$ are *dependent*	iff	α, β are not independent

with $\mathit{enabled} : S \to \mathcal{P}(\mathfrak{S})$ and $\mathfrak{S}$ being the set of all statements (we regard a statement as the subset of those global transitions T in which a specific local transition is taken).

So the different combinations of their interleaving have the same effect. POR tries to select only few of the interleavings having the same result. This is done by choosing in each state s a subset $\mathit{ample}(s) \subseteq \mathit{enabled}(s)$, called the *ample set* of s in [11]. The choice of $\mathit{ample}(s)$ must meet the conditions C0 to C3 listed in Table 2. C3' is a sufficient condition for C3 and can be checked locally in the current state. Since SPIN is an on-the-fly model checker, it uses C3'.

If these conditions are met, then the original graph G and the partial order reduced graph G' are *stuttering equivalent*: For each path $\pi \in G$ there exists a path $\pi' \in G'$ (and vice versa) such that π and π' are stuttering equivalent (cf. [11] and [2]). In our special case of NPCs, two paths π and π' are stuttering equivalent ($\pi \sim_{\mathrm{st}} \pi'$) iff they make the same amount of progress.

3.2 Correctness of $\mathrm{DFS}_{\mathrm{FIFO}}$ in Combination with POR

For proving the correctness of $\mathrm{DFS}_{\mathrm{FIFO}}$ with POR activated, we have to look at the conditions for POR first. C3' no longer implies C3 if $\mathrm{DFS}_{\mathrm{FIFO}}$ is used: Since a large part of the stack gets lost by postponing the traversal at *progresses* (progress states or progress transitions), progress cycles are not detected. To guarantee C3 for progress cycles being traversed, we traverse all

Table 2. Constraints on $ample(s)$

C0: *Emptiness*	$ample(s) = \emptyset \Leftrightarrow enabled(s) = \emptyset$
C1: *Ample decomposition*	No statement $\alpha \in \mathfrak{S} \backslash ample(s)$ that is dependent on some statement from $ample(s)$ can be executed in the original, complete graph after reaching the state s and before some statement in $ample(s)$ is executed.
C2: *Invisibility*	$ample(s) \neq enabled(s) \implies \forall \alpha \in ample(s) : \alpha$ is *invisible*, which means that α is not a progress transition, or, when progress states are being used, that α does not change $np\_$.
C3: *Cycle closing condition*	If a cycle contains a state s s.t. $\alpha \in enabled(s)$ for some statement α, it also contains a state s' s.t. $\alpha \in ample(s')$.
C3': *NotInStack*	$\alpha \in ample(s)$ and $\alpha(s)$ is on the stack $\Rightarrow ample(s) = enabled(s)$

pending transitions when we are about to destroy the stack by making progress. So for each state s we fulfill the condition: $(\exists \alpha \in ample(s) : \alpha$ is visible$) \implies (ample(s) = enabled(s))$. This is equivalent to C2.

Note. When progress states are being used, C2 is not sufficient to guarantee C3 in the special case of cycles that solely contain progress states (e.g., as in Figure 8). Several solutions are possible: Firstly, we can alter C2 to C2': $(\exists \alpha \in ample(s) :$ $\alpha(s)$ is a progress state$) \implies (ample(s) = enabled(s))$. Secondly, these cycles might be avoidable by weak fairness (which is combinable with our algorithm) if they are caused by one process remaining in its progress state throughout the cycle. Thirdly, we can guarantee by hand that these long sequences of progress states never occur, e.g., by forcing quick exit from progress states (similarly to Figure 7). But we favor using progress transitions anyway, which is once more the simplest and most efficient solution.

If DFS_{FIFO} detects a cycle on the stack, it has already found an NPC and is finished. Hence we no longer need C3', C2 suffices to fulfill C3. This fact helps in the following proof.

Theorem 2. *DFS_{FIFO} in combination with POR finds an NPC if one exists and otherwise outputs that no NPC exists. An NPC is found at the smallest depth w.r.t. progress, i.e., after the smallest number (L_0) of progresses that have to be traversed.*

Proof. Partial order reducing the graph G does not create new NPCs. Hence DFS_{FIFO} still does not output false negatives.

To prove completeness, let there be an NPC in G. If $L < L_0$, all paths leading to an NPC are pruned before the NPC is reached.

Let $L = L_0$ and π be a counterexample in G_{L_0}. C3' is unnecessary for DFS_{FIFO}. C0, C1 and C2 are independent of the path leading to s. Therefore $ample(s)$ can be determined independently of the path leading to s. So all

conditions C0, C1, C2 and C3 are met and $ample(s)$ is not influenced by the differing traversal order. Hence stuttering equivalence is preserved. Thus the reduced graph from G_{L_0} that DFS_{FIFO} with POR constructs also has an infinite path with exactly L_0 progresses like π, i.e., an NPC. Theorem 1 proves that after L_0 progresses an NPC is found by DFS_{FIFO} in combination with POR.

3.3 Comparison

Now we can do an overall comparison between SPIN's NPC checks and DFS_{FIFO}, both with POR activated (pros for DFS_{FIFO} are marked with +):

+ DFS_{FIFO} avoids all redundancies and therefore saves time and memory and enables stronger POR.
+ The use of progress transitions instead of progress states is possible, spawning several advantages:
  - The faithful modeling not only simplifies the basic algorithms, but also the application of POR: The visible transitions are exactly the progress transitions, and $\pi \sim_{\text{st}} \pi'$ iff $|\pi|_p = |\pi'|_p$. That is why progress transitions are the easiest solution to get by with C2 instead of C2'.
  - Only one of the originally two local transitions is now visible, i.e., we definitely have fewer visible global transitions.
  - In certain situations, this difference in the number of visible global transitions can get very large: Figure 9 shows that the global automaton has far more visible transitions when progress states are used instead of progress transitions. Consequently, also the ample sets strongly differ in size. The visible transitions are marked with + and -, the ample sets with circles.
+ The constraint C3' becomes unnecessary.
– To obtain an error path from the initial state to the NPC, an additional basic DFS is necessary, but this requires less resources than the main check.
+ A shortest (w.r.t. progress) error path can be given, which is often easier to understand and more revealing than longer paths.
+ By avoiding progress as much as possible, DFS_{FIFO} exhibits an efficient NPC search heuristic: In practice, NPCs often occur after only few progresses. Additionally, by avoiding progress as much as possible, its visibility weakens POR just as much as necessary. Since the time and memory requirements of DFS_{FIFO} and the basic DFS are about the same, the performance of our NPC check is roughly the same as for a safety check if POR stays about as strong as for safety checks.
+ Our new NPC check is a more direct method. This is in line with SPIN's paradigm of choosing the most efficient and direct approach and eases modifications, such as improvements, additional options and extensions.
+ It might be possible to improve POR: For finding NPCs, we only need to distinct $|\pi|_p = \infty$ from $|\pi|_p < \infty$ for an infinite path π. Therefore a stronger reduction that does not guarantee stuttering equivalence is sufficient, as long as at least one NPC is preserved.

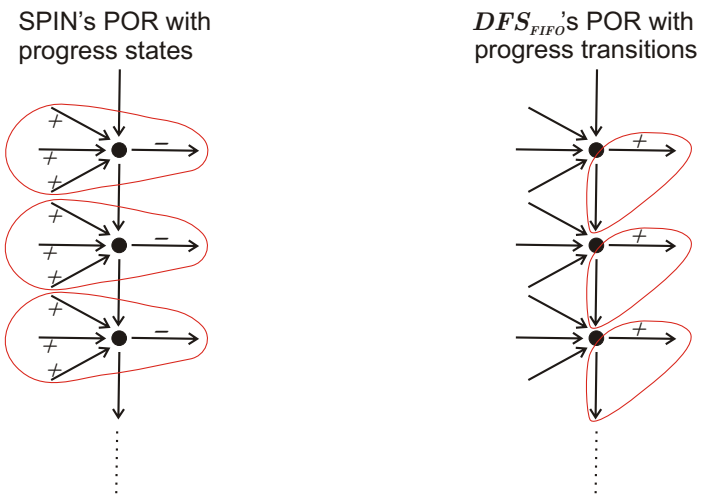

Fig. 9. Smaller ample sets for progress transitions

Note. We can also compare our $\mathrm{DFS}_{\mathrm{FIFO}}$ with SPIN's former NPC check. The old check used the NDFS directly (see [5]). [8] explains that this algorithm is not compatible with POR because of condition C3. The authors of the paper "do not know how to modify the algorithm for compatibility with POR" and suggest the alternative SPIN is now using (cf. Section 1.1). But $\mathrm{DFS}_{\mathrm{FIFO}}$ can be regarded as such modification of SPIN's old NPC check: the state space creation and the search for an NPC are combined, and C3 is reduced to C2.

4 Closure

4.1 Conclusion

Instead of separately constructing the state space and searching for NPCs, as SPIN does, $\mathrm{DFS}_{\mathrm{FIFO}}$ performs both at the same time. To be able to avoid a nested search, $\mathrm{DFS}_{\mathrm{FIFO}}$ postpones traversing progress for the $(L+1)$-th time until the combined state space creation and NPC check for the subgraph G_L is finished. Then $\mathrm{DFS}_{\mathrm{FIFO}}$ retrieves the postponed progresses and continues with $G_{L+1} \setminus G_L$. When an NPC is found or the complete graph is built, $\mathrm{DFS}_{\mathrm{FIFO}}$ terminates. $\mathrm{DFS}_{\mathrm{FIFO}}$ is a more direct NPC check than SPIN's method, with no redundancy, and enabling an efficient search heuristic, better counterexamples, the use of progress transitions, stronger POR and facilitation of improvements. With these enhancements, the verification by NPC checks becomes more efficient. As trade-off, $\mathrm{DFS}_{\mathrm{FIFO}}$ does not deliver an error path from the initial state to the NPC anymore, only the NPC itself. For a complete error path, an additional basic DFS is required - whose cost is, however, negligible.

4.2 Future Work and Open Questions

Having proved that DFS_{FIFO} is correct and combinable with POR, we can now move from these important theoretical questions to the next step of implementing the algorithm. Thereafter, we will analyze DFS_{FIFO}'s performance. Because of the mentioned advantages, we are optimistic that DFS_{FIFO} will strongly improve NPC checks in practice. Section 3.3 posed the open question whether POR can be further strengthened for our NPC checks by weakening stuttering equivalence to a constraint that solely preserves NPC existence. Solving this question might further speed up our NPC check.

References

1. E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer (STTT)*, 2:279–287, 1999. 14
2. Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, third printing, 2001 edition, 1999. 14
3. Yifei Dong, Xiaoqun Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Oleg Sokolsky, Eugene W. Stark, and David S. Warren. Fighting livelock in the i-protocol: a comparative study of verification tools. In *TACAS'99, LNCS*, pages 74–88. Springer, 1999. 2, 11
4. David Faragó. Model checking of randomized leader election algorithms. Master's thesis, Universität Karlsruhe, 2007. 13
5. Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series, 1992. 17
6. Gerard J. Holzmann. *The SPIN Model Checker: primer and reference manual*. Addison Wesley, first edition, 2004. 2
7. Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *Proceedings of the Formal Description Techniques 1994, Bern, Switzerland*, pages 197–211. Chapman & Hall, 1994. 14
8. Gerard J. Holzmann, Doron Peled, and M. Yannakakis. On nested depth-first search. In *Proceedings of the Second SPIN Workshop. Aug. 1996. Rutgers Univ., New Brunswick, NJ.*, pages 23–32. American Mathematical Society. DIMACS/32, 1996. 2, 17
9. Syed M. S. Islam, Mohammed H. Sqalli, and Sohel Khan. Modeling and formal verification of DHCP using SPIN. *IJCSA*, 3(2):145–159, 2006. 2
10. Moataz Kamel and Stefan Leue. Formalization and validation of the general inter-orb protocol (GIOP) using PROMELA and SPIN. In *Software Tools for Technology Transfer*, pages 394–409. SpringerVerlag, 2000. 2
11. Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *6th International Conference on Computer Aided Verification, Stanford, California*, 1994. 14