3. Partial-Order Reduction in Presence of Rendez-vous Communications with Priority Choice and Weak Fairness

This chapter is a revised version of:

Partial Order Reduction in Presence of Rendez-vous Communications with Priority Choice and Weak Fairness

Dragan Bošnački *
Dept. of Computing Science, Eindhoven University of Technology
PO Box 513, 5600 MB Eindhoven, The Netherlands
fax: +31 40 246 3992, e-mail: dragan@win.tue.nl

Abstract. If synchronizing (rendez-vous) communications are used in Promela models, the priority choice feature (unless construct) and the weak fairness algorithm are not compatible with the partial order reduction algorithm used in Spin’s verifier. After identifying the wrong partial order reduction pattern that causes both incompatibilities, we give solutions to these two problems. To this end we propose corrections in the identification of the safe statements for partial order reduction and, as an alternative, we discuss corrections of the partial order reduction algorithm.

1 Introduction

The issue of fairness is an inherent and important one in the study of concurrency and nondeterminism, in particular in the area of verification of concurrent systems. Since fairness is used as generic notion there is a broad taxonomy of fairness concepts. In this chapter we confine our attention to the notion of weak fairness on the level of processes which is implemented in the Spin verifier. This means that we require that for every execution sequence of the concurrent program which is a composition of several processes, if some process becomes continuously enabled at some point of time (i.e. can always execute some of its statements), then at least one statement from that process will eventually be executed. This kind of fairness is most often associated with mutual exclusion algorithms, busy waiting, simple queue-implementations of scheduling, and resource allocation. Weak fairness will guarantee the correctness of statements like eventually entering the critical region for every process which is continuously trying to do that (in the mutual exclusions) or eventually leaving the waiting queue for each process that has entered it (in the scheduling) [10].

Partial order reduction is one of the main techniques that are used to alleviate the problem of state space explosion in the verification of concurrent systems.

* On leave from the Institute of Informatics, Faculty of Natural Sciences and Mathematics, University “Sts. Cyril and Methodius”, Skopje, Macedonia. Supported by EC ESPRIT LTR Project No. 23498 (VIRES).
POR in Presence of Rendez-vous

[20, 11, 14, 16] and it is indeed one of Spin’s main strengths. The idea is, instead of exploring all the execution sequences of a given program, to group them in equivalence classes which consist of interleavings of independent program statements. Then only representatives for each equivalence class are considered. In practice this is realized such that from each state only a subset of the executable statements are taken.

Combining the algorithms for model-checking under weak fairness with partial order reduction is a prerequisite for the verification of many interesting properties to be feasible in practice. However, it was discovered [7] that the two algorithms are not compatible when rendez-vous communications occur in the Promela models. As a result, in the present implementation of Spin (version 3.3 and later) the combination of weak fairness with partial order reduction when rendez-vous are used in the models is not allowed.

Another problem with Spin’s partial order reduction in presence of rendez-vous occurs when the unless construct is used in the Promela models. The combination of these three Spin features is also currently forbidden. As unless is a natural way to represent exceptions, and the latter gains more and more popularity in modern programming languages (e.g. Java), this incompatibility can be a serious drawback.

Interestingly, it turns out that both incompatibilities are caused by exactly the same pattern of wrong partial order reduction. After pointing out this incorrect reduction pattern we propose solutions to the problems with fairness and unless. For both cases we discuss two kinds of solutions, classified according to the two different phases of the verification in which they are implemented. The first kind corrects the identifications of so called safe statements for the partial order reduction algorithm. The marking of statements as safe is done during the compilation of the Promela model, so we call these solutions static. The second kind are the dynamic solutions which are applied during the exploration of the state space and are in fact corrections of the partial order reduction algorithm.

In the next section we give the necessary preliminaries for the rest of the chapter. Section 3 is devoted to partial order reduction and the concrete algorithm that is used in Spin. In Section 4 we discuss the problem with the unless construct and give some solutions to overcome it. Section 5 deals with the Spin’s weak fairness algorithm. We first present the algorithm and show its correctness. After locating the problem and comparing it to the priority choice (unless) case, we again propose both static and dynamic solutions. The last section is a summary with some considerations concerning future work.

2 Preliminaries

In this section, following [14] and [6], we give the semantics of Promela programs (models) and their verification in terms of finite labeled transition systems.

We represent the programs as collections of processes. The semantics of the process $P_i$ can be represented as a labeled transition system (LTS). An LTS
is a quadruple $\langle S_i, \hat{s}_i, \tau_i, L_i \rangle$, where $S_i$ is a finite set of states, $\hat{s}_i \in S_i$ is a distinguished initial state, $L_i$ is a finite set of program statements (labels), and $\tau_i : S_i \times L_i \rightarrow 2^{S_i}$ is a transition function. The transition function induces the set $R_i \subseteq S_i \times L_i \times S_i$ of transitions. Every transition in $R_i$ is the result of an execution of a statement from the process, and $(s_i, a, s'_i) \in R_i$ iff $s'_i \in \tau_i(s_i, a)$.

We introduce a function $\text{Label}$ that maps each transition to the corresponding statement. For a statement $a$, with $\text{Pid}(a)$ we denote the process to which $a$ belongs. (If two syntactically identical statements belong to different processes we consider them as different.) A statement $a$ is enabled in some state $s \in S_i$ iff there exists $s' \in S_i$ such that $s' \in \tau(s, a)$. In this case we also say that the transition $s \xrightarrow{a} s'$ is enabled in $s$. (The enabledness (executability) of a given statement is obtained from the process specification according to rules that we do not consider here.) $\text{En}(a)$ denotes the set of states in which $a$ is enabled. Given a state $s$ and process $p$ we say that $p$ is enabled in $s$ (we write $s \in \text{En}(p)$) if there is a statement $a$ such that $\text{Pid}(a) = p$ and $s \in \text{En}(a)$.

Now we can define the semantics of the program $P$ that corresponds to the concurrent execution of the processes $P_i (1 \leq i \leq N)$ as an LTS which is a product of the labeled transition systems corresponding to the component processes. In the definition we pay a special attention to the transitions generated by rendez-vous communications. We model each such communication as an atomic sequence of a rendez-vous send statement $a$ and a corresponding rendez-vous receive statement $a'$. The statements $a$ and $a'$ generate the consecutive transitions $t$ and $t'$, respectively, such that $\text{Label}(t) = a$ and $\text{Label}(t') = a'$. To preserve the atomicity of the execution, it is required that $t'$ is the only outgoing transition from the intermediate global state obtained after the execution of $t$. The product LTS $(S, \hat{s}, \tau, L)$ consists of:

- state space $S = \prod_{1 \leq i \leq N} S_i$, i.e., the Cartesian product of the state spaces $S_i$,
- initial state $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_N)$,
- $L = \bigcup_{1 \leq i \leq N} L_i$, i.e., the set of statements is the union of the statement sets of the components,
- and the transition function defined as:
  1. if $a$ is not a rendez-vous statement, then $(s_1, \ldots, s_k, \ldots, s_N) \in \tau(s_1, \ldots, s_k, \ldots, s_N, a)$ if $s'_k \in \tau_k(s_k, a)$
  2. if $a$ is a rendez-vous send, and there is a rendez-vous receive statement $a'$, such that $\text{Pid}(a') \neq \text{Pid}(a)$, $s'_k \in \tau_k(s_k, a)$ and $s'_i \in \tau_i(s_i, a')$, then
     a) $(s_1, \ldots, s'_k, \ldots, s_i, \ldots, s_N) \in \tau(s_1, \ldots, s_k, \ldots, s_i, \ldots, s_N, a)$
     b) $(s_1, \ldots, s'_k, \ldots, s'_i, \ldots, s_N) \in \tau(s_1, \ldots, s_k, \ldots, s'_i, \ldots, s_N, a')$
     c) There is no action $a'' \neq a'$ such that $(s_1, \ldots, s'_k, \ldots, s_i, \ldots, s_N) \in \text{En}(a'')$, i.e., no other statement can be interleaved with the execution of receive;

A popular way to represent requirements on the program is by a linear temporal logic (LTL) formula [9]. In Spin, next-time-free LTL is used, which means that formulae may contain only boolean propositions on system states, the boolean
operators $\wedge$, $\lor$, $!$ (negation), and the temporal operators $\Box$ (always), $\Diamond$ (eventually) and $U$ (until). For verification purposes the LTL formulae are translated into Büchi automata.

A Büchi automaton is a tuple $B = (\Sigma, S, \rho, 0, F)$, where:

- $\Sigma$ is an alphabet,
- $S$ is a set of states,
- $\rho: S \times \Sigma \to 2^S$ is a transition function,
- $0 \in S$ is the initial state,
- and $F \subseteq S$ is a set of designated states called acceptance states.

A run of $B$ over an infinite word $w = a_1a_2\ldots$, is an infinite sequence $s_0s_1\ldots$ of states, where $s_0$ is the initial state $0$ and $s_i \in \rho(s_{i-1}, a_i)$, for all $i \geq 1$. A run $s_0s_1\ldots$ is accepting if for some $s \in F$ there are infinitely many $i$’s such that $s_i = s$. The word $w$ is accepted by $B$ if there is an accepting run of $B$ over $w$.

The transitions of the Büchi automaton that is obtained from the formula are labeled with boolean propositions over the global system states of the LTS corresponding to the program, i.e., the alphabet $\Sigma$ is a set of boolean propositions.

In order to prove the satisfaction of the LTL formula by the program $P$, we further define the synchronous product of the LTS $(S_P, 0_P, s_1, \tau_P, L_P)$ corresponding to $P$ and the Büchi automaton $A = (\Sigma, S_A, \rho, 0_A, F_A)$ obtained from the negation of the LTL formula, to be an LTS extended with acceptance states, i.e., extended LTS (XLTS) $(S, 0, s, \tau, L, F)$,\(^1\) with:

- state set $S = S_P \times S_A$,
- initial state $0 = (0_P, 0_A)$,
- transition function $\tau: S \times L \to 2^S$ defined as $(s_{2P}, s_{2A}) \in \tau(s_{1P}, s_{1A}, a)$ iff $s_{2P} \in \tau_P(s_{1P}, a)$ and there is a proposition $p \in \Sigma$ such that $s_{2A} \in \rho(s_{1A}, p)$ and $p$ is true in $s_{1P}$,
- set of statements $L = L_P$,
- and a set of designated acceptance states $F = \{(s_P, s_A) | s_A \in F_A\}$, i.e. we declare as acceptance states the states with second component belonging to the acceptance set of the Büchi automaton $A$.

Unless stated differently, for the rest of the chapter we fix the XLTS $T$ to be the tuple $(S, 0, s, \tau, L, F)$ as defined above. With $R$ we denote the set of transitions of $T$.

An execution sequence or path is a finite or infinite sequence of subsequent transitions, i.e., for $s_i \in S$, $a_i \in L$, the sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots$ is an execution sequence in $T$ iff $s_i \xrightarrow{a_i} s_{i+1} \in R$ for all $i \geq 0$. An infinite execution sequence is said to be accepting iff it starts in the initial state $0$ and there is an acceptance

\(^1\)Although the proliferation of different formal models (LTS, Büchi automata, extended LTS) that are used to represent the semantics and the state space might seem unnecessary, we use three different formal concepts in order to follow more closely [14] and [6], so that we will be able to reuse most of the results from these papers in a seamless way.
state \( s \in F \) that occurs infinitely many times in the sequence. A finite execution sequence \( c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} s_n \) \((n \geq 1)\) is a \textit{cycle} iff the start and end states coincide, i.e. \( s_0 = s_n \). Given a finite or infinite execution sequence \( \sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \), a process \( P_i \), \( 1 \leq i \leq N \), and a state \( s_j \) from the execution sequence, we say that \( P_i \) is \textit{executed} in \( \sigma \) in \( s_j \) iff \( \text{Pid}(a_j) = i \). A state \( s \) is \textit{reachable} iff there exists a finite execution sequence that starts at \( s \) and ends in \( s \). A cycle \( c \) is \textit{reachable} iff there exists a state in \( c \) which is reachable. A cycle \( c \) is an \textit{acceptance cycle} if it contains at least one acceptance state.

The satisfaction of the formula by the program \( P \) can now be proven by showing that there are no accepting execution sequences of the extended LTS \( T \). On the other hand, the existence of accepting execution sequences means that the formula is not satisfied. From the definition of Büchi automata and extended LTS and following the reasoning from [6], for instance, it is straightforward to conclude that the extended LTS has an accepting execution sequence iff it has some state \( s \in F \) that is reachable from the initial state and reachable from itself (in one or more steps) [6]. Thus, we have to look for reachable acceptance cycles in the XLTS \( T \). In the sequel we call an XLTS also a \textit{state space}.

\section{Partial Order Reduction}

In this section we give a brief overview of the partial order reduction (POR) algorithm by Holzmann and Peled [14], that is considered throughout the chapter. This algorithm is also implemented in Spin. We start by rephrasing some definitions from [14].

The basic idea of the reduction is to restrict the part of the state space that is explored by the DFS, in such a way that the properties of interest are preserved. To this purpose, the independence of the checked property from the possible interleaving of statements is exploited. More specifically, two statements \( a, b \) are allowed to be permuted precisely then, if for all sequences \( v, w \) of statements: if \( vabw \) (where juxtaposition denotes concatenation) is an accepted behavior, then \( vbaw \) is an accepted behavior as well. In practice, sufficient conditions for such permutability are used that can be checked locally, i.e., in a state. For this, a notion of “concurrency” of statements is used that captures the idea that transitions are contributed by different, concurrently executing processes of the system.

We first introduce some additional terminology. Without loss of generality we assume that the transition function \( \tau \) of the LTS representing the program is deterministic, i.e., the set \( \tau(s, a) \) consists of at most one element, for any state \( s \in S \) and any statement \( a \in L \). For \( q \in En(a) \), let \( a(q) \) be the state which is reached by executing \( a \) in state \( q \). Concurrent statements (i.e. statements with different \( \text{Pids} \)) may still influence each other’s enabledness, whence it may not be correct to only consider one particular order of execution from some state. The following notion of independence defines the absence of such mutual influence. Intuitively, two statements are independent if in every state where they are both
enabled, they cannot disable each other, and are commutative, i.e., the order of their execution makes no difference to the resulting state.

**Definition 1.** The statements $a$ and $b$ are independent iff for all states $q$ such that $q \in En(a)$ and $q \in En(b)$,
- $a(q) \in En(b)$ and $b(q) \in En(a)$, and
- $a(b(q)) = b(a(q))$.

**Statements that are not independent are called dependent.**

Note that $a$ and $b$ are trivially independent if $En(a) \cap En(b) = \emptyset$. An example of independent statements are assignments to or readings from local variables, executed by two distinct processes.

Also note that the statements $a$ and $b$ are considered to be independent even if $a$ can enable $b$ (and vice versa). The main requirement is that the statements do not disable each other. This is unusual in a sense, because in the literature a more strict definition prevails that does not allow that a statement can enable another statement (e.g. [20, 11]). The advantage of the subtlety in Definition 1 is that it ensures a greater set of independent statements than the “classical” definition and consequently a better reduction of the state space. However, we must be careful with this, because as we will see later, this feature is closely connected with the incompatibilities that we are discussing in this chapter.

Another reason why it may not be correct to only consider only one particular order of execution from state $s$ of two concurrent statements $a$ and $b$ is that the difference between the intermediate states $a(s)$ and $b(s)$ may be observable in the sense that it influences the property to be checked. For a given proposition $p$ that occurs in the property (an LTL formula), and a state $s$, let $p(s)$ denote the boolean value of the proposition $p$ in the state $s$. Then, $a$ is nonobservable iff for all propositions $p$ in the property and all states $s \in En(a)$, we have $p(s) = p(a(s))$. The statement $a$ is said to be safe if it is nonobservable and independent from any other statement $b$ for which $Pid(b) \neq Pid(a)$.

In the rest of this section we describe in a rather informal way the partial order algorithm from [14]. For the full details about the algorithm we recommend the original references [14, 16].

The reduction of the search space is effected during the DFS, by limiting the search from a state $s$ to a subset of the transitions that are enabled in $s$, the so-called ample set. Such an ample set is formed in the following way: If there is a process $P_i$ who can potentially execute only safe statements in $s$, and for all transitions $s \xrightarrow{a} s'$ that are labeled with statements from $P_i$ (i.e. $Pid(a) = i$) the state $s'$ is not on the DFS stack, then the ample set consists of all the transitions from this process only, i.e. all transitions $t$ from $s$ such that $Pid(Label(t)) = i$. Otherwise, the ample set consists of all enabled transitions in $s$.

If we now introduce the notion of ample process set, consisting of the processes that have a transition in the ample set, then the reduced DFS algorithm is obtained by replacing in the standard DFS exploration algorithm (Fig. 1) line 4 by the line

\[\text{See for more details [14].}\]
for each process \( i \) in ample process set do

Obviously the ample process set consists of either only one or all the processes in the system. In the latter case there is no reduction and the reduced search algorithm behaves as the standard DFS, for this particular invocation.

```
proc dfs(s)
  add s to Statespace
  /* for each successor s' of s do */
  for each process \( i = 1 \) to \( N \) do
    nxt = all transitions enabled in s with Pid(t)=i
    for all t in nxt do
      \( s' = \) successor of s via t
      if \( s' \) not in Statespace then
        dfs(s')
      fi
    od
  od
end
```

Fig. 1. Standard depth first search algorithm.

It can be proven \([14, 16]\) that one can use the reduced state space (XLTS) \( R(T) \), obtained as discribed above, instead of the original XLTS \( T \) to check any program property which is stated as an LTL formula. In \([14, 16]\) it is shown that

**Theorem 1** (\([14, 16]\)). There exists a reachable acceptance cycle in \( T \) iff there exists a reachable acceptance cycle in \( R(T) \).

The acceptance cycle in the reduced state space is detected by the cycle-check algorithm in Spin that we consider in more detail in Section 5.1.

The condition that all transitions from the ample set must end out of the DFS stack, the so-called “cycle proviso”, ensures that a statement that is constantly enabled, cannot be “forgotten” by leaving it outside the ample set in a cycle of transitions.

While the cycle proviso is clearly locally checkable during a DFS, the condition that an enabled statement is safe is not, as the definition of safety requires independence from any concurrent statement. A sufficient condition for safety of a statement \( a \) that can be checked locally is that \( a \) does not touch any global variables or channels. Indeed, it is this condition that is implemented in Spin.

However, it will turn out that one solution for our incompatibility problems will be to refine this safety criterion.

4 **The unless Construct**

The unless construct is a mean for modeling exception handling routines and priority choices. Its syntax is \( stmt \) unless \( stmt \). The first (left-hand) statement
is called normal or main, while the second (right-hand) is the escape statement. Semantically, the executability of the normal statement depends on the executability of the escape sequence. The escape sequence has higher priority than the normal statement, which means that the normal statement will be executed only if the escape statement is not executable. Otherwise the escape statement is executed and the normal statement is ignored (skipped). This dependence between the two statements of unless causes problems when the partial order reduction is used and the escape statement is a rendez-vous communication.

```promela
chan c = [0] of {bit}

active proctype A()
{
    skip; c?1;
}

active proctype B()
{
    assert(false) unless c!1;
}
```

**Fig. 2.** Motivating example for unless statement.

Let us consider the motivating Promela example given in Figure 2.\(^4\) (In the sequel we assume that the reader is familiar with Promela.) Suppose that both A and B are in their starting points, i.e. A is trying to execute its skip statement, while B is attempting to do its unless statement. Obviously the higher priority rendez-vous send offer c!1 issued by B cannot find a matching receive, so the verifier should detect the assertion violation `assert(false)`.\(^5\) However, in the reduced search this is not detected, because of the incorrect partial order reduction. The problem with the reduction occurs because the skip statement is not safe anymore. Namely, the criterion that a statement is safe if it does not affect any global objects is no longer true. Because the executability

---

\(^3\) In general, both statements can be sequences of Promela statements. Also the unless construct can be nested. The results from this chapter can be extended in a straightforward way for this general case.

\(^4\) The example is distilled from a model made in the discrete time extension of Spin DTSpin [2]. The model was written by Victor Bos, who first drew our attention to the possible problems with the unless statement.

\(^5\) Strictly speaking in this example we are considering a safety property that is not expressed as an LTL formula. The equivalent formulation of the property in LTL can be done in a straightforward way and the partial order reduction will fail because of the same reason as in the present case. We use this version of the example for the sake of simplicity.
of the rendez-vous statement \( c?1 \) can be changed only because of the change of the location in process \( A \) (program counter), no statements are unconditionally globally independent according to the Definition 1.

\[
\text{assert(false)}
\]

\[
\text{assert(false)}
\]

\[
\text{c!1 | c?1}
\]

\[
\text{c!1 | c?1}
\]

**Fig. 3.** Interdependence between the **unless** construct and a statement which is safe in absence of **unless**.

The reason is depicted in Fig. 3. In the starting state described above the rendez-vous send \( c!1 \) is disabled, but with the execution of \text{skip} it becomes enabled. This means that \text{skip} has indirectly disabled \text{assert(false)} which was enabled in the starting state. In that way \text{skip} and \text{assert(false)} are not independent according to the Definition 1, because \text{skip} disables \text{assert(false)}.

The problem can be solved both statically in compile time or dynamically during the exploration of the state space. The dynamic solution consists of checking whether in a given state there is a disabled rendez-vous statement (more precisely, rendez-vous send) which is part of an escape statement and in that case the partial order reduction is not performed in the given state. The drawback of this solution is that it can be time consuming.

The static solution is to simply declare each statement which is followed in the process specification (**proctype**) by a rendez-vous communication (more precisely, by a rendez-vous receive) as unsafe. We use the term **followed** taking into account all the cycles and jumps in the Promela specification. For example, the last statement of the body of an iteration is followed by the first statement of the body. Whether a given statement is followed by a rendez-vous can be checked by inspecting Spin’s internal representation of the Promela program (abstract syntax tree). This can be done during the generation of the C source (**pan.c**) of the special purpose verifier for the program. Thus, the solution does not cause any time overhead during the verification. Its drawback with regard to the dynamic solution is that the reduction can be less effective because of an unnecessary strictness. It can happen that the reduction is unnecessarily prevented even when in the state that is considered by the DFS there is no
POR in Presence of Rendez-vous

disabled rendez-vous send in an unless construct or even there is no statement with an unless construct at all.

5 Fairness

A pattern very similar to the one from the previous section that causes the partial order reduction algorithm to fail in presence of rendez-vous communications, occurs when the weak fairness option is used in the verification. The weak fairness algorithm is also a very instructive example how things can become complicated because of the feature interaction.

5.1 The Standard Nested Depth-First Search (NDFS) Algorithm

The weak fairness algorithm we are going to deal with is an extension of the nested depth first search (NDFS) algorithm by Courcoubetis, Vardi, Wolper and Yannakakis [6] for memory efficient verification of LTL properties. The algorithm is a more efficient alternative to the usual computation of the strongly connected components of the underlying graph for the XLTS. It is also compatible with Spin’s bit-state hashing, which is not the case with the strongly connected components algorithm. We start with a brief overview of the NDFS algorithm given in Figure 4.

The core idea of the algorithm is to extend the standard DFS of the state space in Fig. 1 with a procedure that checks for an acceptance cycle. Thus, whenever the standard DFS is about to retract from an acceptance state, it is interrupted and the cycle-check procedure is called. The procedure is again a DFS which is started with an acceptance state as a root (seed). If the root state is matched within the cycle-check procedure, an acceptance cycle is reported and the algorithm is stopped. Otherwise, the standard DFS is resumed from the point it has been interrupted. If no cycle is found, then the property is successfully verified.

We need to work with two copies of the state space in order to ensure that the second DFS does not fail to detect a cycle by cutting the search because it has encountered a state visited already by the first DFS. To distinguish between states belonging to different copes we extend the state with one bit denoting the state space copy.

The following theorem from [6] establishes the correctness of the algorithm.

Theorem 2 ([6]). Given an XLTS $T$, when started in the initial state $\hat{s}$, the algorithm in Fig. 4 reports a cycle iff there is a reachable acceptance cycle in $T$.

Note 1. An important feature of the NDFS algorithm is that any state in the second copy of the state space is visited in at most one of the calls to dfs2. This is due to the characteristic position of the call of the cycle-check procedure dfs2, namely, immediately before the recursion retracts from an acceptance state. As shown in [6], this ensures that the part of the second copy of the state space that has been explored by the previous calls of dfs2 can be reused in the current call of the cycle-check procedure.
proc dfs1(s) \\
   add {s,0} to Statespace \\
   /* for each successor s' of s do */ \\
   for each process i = 1 to N do \\
      nxt = all transitions enabled in s with Pid(t)=i \\
      for all t in nxt do \\
         s' = successor of s via t \\
         if {s',0} not in Statespace then \\
            dfs1(s') \\
         fi \\
      od \\
   od \\
   if acceptance(s) then \\
      seed:={s,1}; \\
      dfs2(s) \\
   fi \\
end \\

proc dfs2(s) /* the nested search */ \\
   add {s,1} to Statespace \\
   /* for each successor s' of s do */ \\
   for each process i = 1 to N do \\
      nxt = all transitions enabled in s with Pid(t)=i \\
      for all t in nxt do \\
         s' = successor of s via t \\
         if {s',1} not in Statespace then \\
            dfs2(s') \\
         else \\
            if {s',1}=seed then \\
               report cycle \\
            fi \\
         fi \\
      od \\
   od \\
end \\

Fig. 4. Nested depth first search algorithm.
5.2 Description of the Weak Fairness Algorithm

We consider weak fairness with regard to processes, i.e. we say that a given execution sequence is fair if for each process that becomes continuously enabled starting at some point in the execution sequence, a transition belonging to this process is eventually executed. Formally

**Definition 2.** An infinite execution sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \ldots$ is fair iff for each process $P_l, 1 \leq l \leq N$, the following holds: If there exists $i \geq 0$ such that $P_l$ is enabled in $s_j$ for all $j \geq i$, then there are infinitely many $k \geq i$ such that $P_l$ is executed in $s_k$.

A cycle $c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_{n-1}} s_0$ is fair iff whenever a process $P$ is enabled in all states $s_i, 0 \leq i < n$, then $P$ is executed in some state $s_j, 0 \leq j < n$.

When model checking under fairness, we are interested only in fair acceptance runs (sequences). This means that we require the detected cycles to be fair, i.e. each continuously enabled process along the cycle contributes at least one transition to it.

The weak fairness (WF) algorithm we are considering here is by Gerard Holzmann and it is a variant of Choueka’s flag algorithm [13]. The weak fairness algorithm is also implemented in Spin.

The basic idea behind the weak fairness algorithm is to apply the NDFS to an extended state space instead of the original one. The extended state space is designed such that to each fair acceptance cycle from the original state space, there is a corresponding acceptance cycle, which need not be fair, in the extended state space, and vice versa. As a consequence, the model checking can be done on the extended state space instead on the original one.

The extended state space consists of $N + 2$ copies of the original state space, where $N$ is the number of processes. To distinguish between the different copies of a same state $s$, we extend each state with a counter component whose value denotes the copy of the state space the state $s$ belongs to. Intuitively, when the algorithm operates in a particular copy, this means that it has reached a certain phase. Our goal is to detect a cycle that:

1. contains at least one state which is an acceptance state in the original state space, and
2. along which each process either executes a transition or becomes disabled.

In order to treat the two cases of condition 2 uniformly, we consider that when process $i$ is disabled, it executes a special $\epsilon_i$-transition, where $1 \leq i \leq N$, which does not change the state. The $\epsilon_i$-transitions are labeled with the special statement $\epsilon_i$. This statement differs from all other statements and only changes the state counter component to $i + 1$.

We check the fulfillment of the two conditions above in a sequential manner. To this end the algorithm passes cyclically through all the copies, from 0 through $N + 1$ and back to 0.\(^6\)

\(^6\) As we will see shortly, the NDFS algorithm applied on the extended state space starts in copy 0, while the cycle checks with $\text{dfs2}$ begin in copy $N + 1$. See also the code of the algorithm in Fig. 7.
Schematically, the extended state space is depicted in Fig. 5. The algorithm resides in copy 0 until an acceptance state is encountered, i.e., condition 1 is achieved. When this happens it immediately passes to copy 1. To this end we introduce another type of special transitions, labeled with the special statement $\epsilon_0$. Like $\epsilon_i$-transitions, an $\epsilon_0$-transition too does not change the state, but only the counter (in this case from 0 to 1). The $\epsilon_0$-transitions (statements), however, do not belong to any process.

Copies 1 to $N$ correspond to each of the processes. Intuitively, when the algorithm operates in copy $i$, $1 \leq i \leq N$, it is waiting for process $i$ to execute a transition (ordinary or $\epsilon$). When in copy $i$ process $i$ executes a transition the algorithm passes to copy $i + 1$. Otherwise, if some other process executes a transition, the algorithm stays in copy $i$.

The cycle check always starts in copy $N + 1$. All the states in this copy are considered as acceptance states in the new extended state space. The most important characteristic of the copy $N + 1$ is that, unlike in the other copies, there are no transitions from itself leading back to it. More precisely, each transition that generates a state in copy $N + 1$ is immediately followed by an $\epsilon_0$-transition to copy 0. This is to avoid detection of acceptance cycles which might be closed by staying inside copy $N + 1$ and which might not satisfy conditions 1 and 2.
From the state space structure one can see that each cycle passing through copy \( N + 1 \) must also pass through all the other copies. Therefore, such a cycle satisfies conditions 1 and 2 and corresponds to a fair acceptance cycle in the original state space.

In the sequel we give more formal treatment of the intuitive picture given above. We begin by defining the extended state space:

**Definition 3.** Given an XLTS \( T = (S, s, \tau, L, F) \) and processes \( P_1, \ldots, P_N \), \( N \geq 2 \), we define its (weakly) fair extension \( F(T) \) to be the smallest XLTS \( (S_f, \hat{s}_f, \hat{\tau}_f, L \cup \{\epsilon_0, \epsilon_1 \ldots \epsilon_N\}, F_f) \), where \( L \cap \{\epsilon_0, \epsilon_1 \ldots \epsilon_N\} = \emptyset \), satisfying:

- \((\hat{s}, 0) \in S_f\)
- \(1. \text{ if } (s, 0) \in S_f, \ s \notin F, \text{ and } s' \in \tau(s,a) \text{ for some } a \in L, \text{ then } (s', 0) \in \tau_f((s, 0), a)\),
- \(2. \text{ if } (s, 0) \in S_f, \ s \in F, \text{ then } (s, 1) \in \tau_f((s, 0), \epsilon_0)\),
- \(3. \text{ if } (s, C) \in S_f, 1 \leq C \leq N, \ s' \in \tau(s,a), \text{ and } \text{Pid}(a) \neq C, \text{ then } (s', C) \in \tau_f((s, C), a)\),
- \(4. \text{ if } (s, C) \in S_f, 1 \leq C \leq N, \ s' \in \tau(s,a), \text{ and } \text{Pid}(a) = C, \text{ then } (s', C + 1) \in \tau_f((s, C), a)\),
- \(5. \text{ if } (s, C) \in S_f, 1 \leq C \leq N, \text{ and } P_C \text{ is not enabled in } s, \text{ then } (s, C + 1) \in \tau_f((s, C), \epsilon_C)\),
- \(6. \text{ if } (s, N + 1) \in S_f, \text{ then } (s, 0) \in \tau_f((s, N + 1), \epsilon_0)\). \(\hat{s}_f = (\hat{s}, 0)\)
- \(F_f = \{(s, N + 1) | s \in S \} \cap S_f\).
- \(\text{Pid}(\epsilon_0) = 0 \text{ and } \text{Pid}(\epsilon_i) = i, \text{ for all } 1 \leq i \leq N\).

The algorithm which generates and explores the fair extension \( F(T) \) is given in Fig. 6. The pseudo-code is an extension of the standard DFS (Fig. 1) that implements Def. 3. Lines 2 and 3 implement case 2 of the definition (\( \epsilon_0 \)-transitions from copy 0 to copy 1). The recursive call of dfs changes only the counter component. Similarly, lines 5 and 6 implement case 6 (\( \epsilon_0 \)-transitions from copy \( N + 1 \) to copy 0). cases 3, 4 and 5 are implemented in the main part of the algorithm (lines 7–19). The counter increment required in cases 4 and 5 is captured by line 9. If the value of the counter \( C \) does not match the process ID \( i \) then there is no increment, which is in accord with case 3. Further, case 5 (\( \epsilon_i \)-transitions) is implemented with lines 11 and 12. The check for \( C = i \) in line 12 is to ensure that an \( \epsilon_i \)-transition is created only if the ID of the disabled process matches the counter component. Finally, case 0 is also implicitly captured. Namely, if the counter component \( C \) is 0, but \( s \) is not an acceptance state in the original state space, then lines 2 and 3 do not apply, and also there is no increment of the counter component in line 9, because \( C \) is 0 and \( i \) ranges from 1 to \( N \).

An obvious way to check the system under weak fairness is to first generate \( F(T) \) from \( T \), using the algorithm from Fig. 6 and then apply the NDFS algorithm to \( F(T) \). Because of the efficiency reasons we will do these two steps simultaneously, i.e., combine them in the on-the-fly algorithm given in Fig. 7.

The new version of the algorithm is obtained in an obvious way by inserting the lines for cycle check, i.e., applying the standard NDFS scheme from Fig. 4.
to the exploration of $F(T)$ from Fig. 6. For the correctness of the algorithm we have to show the following cycle correspondence between the original and the extended state space:

**Lemma 1.** There exists a reachable fair acceptance cycle in $T$ iff there exists a reachable acceptance cycle in $F(T)$.

**Proof.** Let us denote the sets of transitions of $T$ and $F(T)$ with $R$ and $R_f$, respectively. Let $c = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{n-1}} s_n$ be a reachable fair acceptance cycle in $T$. We first prove by induction that for each state $s$ in $T$, if $s$ is reachable, then $(s, C)$ in $F(T)$, for some $C \geq 0$, is also reachable. The induction is on the length of the shortest path(s) between the initial state $\hat{s}$ and $s$. For length 0 the claim trivially holds, because $\hat{s}$ is the only state which is reachable via a path with length 0 and by Def. 3 the state $(\hat{s}, 0)$ is in $F(T)$. Assume that $s$ is reachable from $\hat{s}$ via path of length $n + 1$, $n \geq 0$. Then there exists a state $s'$ reachable through a path of length $n$, such that $s' \xrightarrow{a} s \in R$, for some $a \in L$. By the induction hypothesis there exists $(s', C')$ in $F(T)$ which is reachable. We will prove that there exists an execution sequence leading from $(s', C')$ to $(s, C)$, $C \geq 0$. We have to consider several cases, taking into account Def. 3:

(i) If $1 \leq C' < N$, then the execution sequence consists of only one transition which is implied by cases 3 and 4 (and because of the transition $s' \xrightarrow{a} s \in R$).

(ii) If $C' = 0$ and $s'$ is not an acceptance state in $T$, then the desired execution sequence (transition) exists because of case 1.
proc dfs1(s,C)
  add {s,C,1} to Statespace
  if C == 0 and acceptance(s) then
    dfs1(s,1) /* epsilon0 move to copy 1 */
  else if C == N+1 then
    dfs1(s,0) /* epsilon0 move to copy 0 */
  else
    for each process i = 1 to N do
      if C == i then C' = C+1 else C' = C fi
      nxt = all transitions t enabled in s with Pid(t)=i
      if nxt = empty then /* epsilon move */
        if {s,C',1} not in Statespace and C == i then dfs1(s,C') fi
      else
        for all t in nxt do
          s' = successor of s via t
          if {s',C',1} not in Statespace then dfs1(s',C') fi
        od
      fi
    od
  fi
  if acceptance((s,C)) then seed:={s,C,1}; dfs2(s,C) fi
end

proc dfs2(s,C) /* the nested search */
  add {s,C,1} to Statespace
  if C == 0 and acceptance(s) then
    dfs2(s,1) /* epsilon0 move to copy 1 */
  else if C == N+1 then
    dfs2(s,0) /* epsilon0 move to copy 0 */
  else
    for each process i = 1 to N do
      if C == i then C' = C+1 else C' = C fi
      nxt = all transitions enabled in s with Pid(t)=i
      if nxt = empty then /* epsilon move */
        if {s,C',1} == seed then report cycle
        else if {s,C',1} not in Statespace and C == i then dfs2(s,C') fi
      else
        for all t in nxt do
          s' = successor of s via t
          if {s',C',1} = seed then report cycle
          else if {s',C',1} not in Statespace then dfs2(s',C') fi
        od
      fi
    od
  fi
end

Fig. 7. Weak fairness (WF) algorithm.
(iii) If \( C' = 0 \) and \( s' \) is an acceptance state in \( T \), then, by case 2, there exists the transition \( (s',0) \xrightarrow{c_0} (s',1) \) and this case reduces to case (ii).

(iv) If \( C' = N + 1 \), by case 6 there exists the transition \( (s', N + 1) \xrightarrow{c_0} (s', 0) \) and the proof boils down to case (ii) or case (iii).

We construct an acceptance cycle in \( F(T) \) that corresponds to \( c \), by unfolding \( c \) according to the definition of \( \tau_f \) (Def. 3). Without loss of generality suppose that we start at \( s_0 \) by mapping it to \((s_0, C) \in F(T)\) for some \( C \geq 0 \), whose existence we just proved. We continue traversing \( c \) and mapping its transitions to transitions in \( F(T) \) according to the definition of \( \tau_f \). The only ambiguity that might occur is resolved such that case 5 from the definition of \( \tau_f \) has priority over case 3. More precisely, suppose we have to extend the obtained path in \( F(T) \) from some state \((s, C') \in S_f \) by mapping a transition \( s \xrightarrow{a} s' \in R \) and the process \( C' \) is disabled (so, \( Pid(a) \neq C' \)). In this case we first extend the path in \( F(T) \) with \((s, C') \leq_c (s, C' + 1) \in R_f \) and after that we continue the mapping of \( s \xrightarrow{a} s' \). As \( c \) is fair, after several steps (bounded by \( N(n + 1) \)) we will arrive at some acceptance state \((s_{a_1}, N + 1) \in F(T)\). According to case 6 of Def. 3, from this state we pass via an \( \epsilon_0 \)-transition to \((s_{a_1}, 0) \). After several steps we will encounter an acceptance state from which we pass via a \( \epsilon_0 \)-transition to a state with counter value 1. The counter is increased until again some acceptance state \((s_{a_2}, N + 1) \) is hit. If \( s_{a_1} = s_{a_2} \), then we are done because we have closed the desired acceptance cycle. Otherwise, we repeat the whole procedure from \((s_{a_1}, N + 1) \). As the number of states in \( c \) is finite, we will inevitably revisit some acceptance state \((s_{a_k}, N + 1) \) and in that way generate the corresponding cycle in \( F(T) \).

For the other direction, observe that each acceptance cycle in \( F(T) \) must contain some state \((s, 0) \), such that \( s \in F \) is an acceptance state in \( T \). This is because in \( F(T) \) from an acceptance state we always pass to a state with counter component 0 (Def. 3, case 6). From Def. 3 it is clear that the only transition from copy 0 is via a \( \epsilon_0 \)-transition (case 2), from some state \((s', 0) \) such that \( s' \in F \). Further, it is obvious that in order to get back to the acceptance state in copy \( N + 1 \), one has to pass through all copies from 1 to \( N \), i.e., each process contributes a transition to the cycle, or is disabled in some state. So, each acceptance cycle in \( F(T) \) can be mapped into a fair acceptance cycle in \( T \) by simply removing the \( \epsilon_0 \)- and \( \epsilon_1 \)-transitions and merging accordingly the states that are connected via these. In the same way we can map any path from the initial state \((\hat{s}, 0) \) to some state \((s, C) \) on the acceptance cycle in \( F(T) \), into a path from \( \hat{s} \) to \( s \) in \( T \), which gives the reachability of the obtained cycle in \( T \).

\[\square\]

As the WF algorithm is in fact the NDFS algorithm applied to \( F(T) \), the correctness of the algorithm is implied by Lemma 1 and Theorem 2, which imply the following theorem

**Theorem 3.** Given an XLTS \( T \), when started in the initial state \((\hat{s}, 0) \) of \( F(T) \), the algorithm in Fig. 7 reports a cycle iff there is a reachable fair acceptance cycle in \( T \).
Note 2. The WF algorithm can be optimized by eliminating $\epsilon_0$ and $\epsilon_i$-transitions. In Spin there are no $\epsilon_0$-transitions. Also part of the $\epsilon_i$-transitions are eliminated and all intermediate states that are produced by the $\epsilon_i$-transitions. However, these optimizations can be regarded as an implementational detail which is not relevant for the discussions in this chapter.

5.3 (In)compatibility with the Partial Order Reduction Algorithm

The Motivating Counter Example. One can regard $F(T)$ as an ordinary LTS by abstracting from the state structure. Thus, it is straightforward to show that $F(T)$ is compatible with the partial order reduction techniques in general ([16,14,11,20,18]), provided that the original set of independent statements in $T$ is adjusted in an appropriate way. More precisely, one has to reduce the set of independent statements because of the $\epsilon_i$-transitions ($1 \leq i \leq N$). Notice that $\epsilon_0$-transitions are not a problem because they are trivially independent of any other action in the system. This is because a $\epsilon_0$-statement is always the only action (transition) from a given state, i.e., it is never simultaneously enabled with any other action (from the Def. 3, cases 2 and 6). The $\epsilon_i$-statements can be dependent with some of the statements that are independent in $T$. In what follows we explain the dependence and show how the compatibility of the WF algorithm with POR in Spin can be repaired by restricting the set of independent statements.

The incompatibility of the weak fairness algorithm with POR (because of rendez-vous communications) was first observed by Dennis Dams on the example in Fig. 8:

In the model from Fig. 8 the LTL formula $\diamond p$, where $p$ is defined as $b == true$, is not valid even under fairness. The crucial role in the incompatibility is played by the statement $x = 0$ in process $C$. Namely, because of this statement, the rendez-vous send $c!1$ from process $B$ is no longer continuously enabled. Since, now there exists a fair cycle formed just by the statements from $A$ and $C$, i.e., $c!0; c?x; x = 0; c!0 \ldots$ along which the process $B$ can be safely ignored because it is not continuously enabled.

However, when partial order reduction was used the verifier incorrectly re-reported that the formula was valid. So, the aforementioned fair cycle formed by the processes $A$ and $C$ was not discovered. The reason for the failure is very similar to the one for the unless statements. Again the same pattern occurs of a wrong partial order reduction because of the incorrect independence relation. But this time the problem is related to the $\epsilon_i$-transitions. Recall that those transitions only change the counter component $C$ when all statements of the process $P$, such that $\text{Pid}(P) = C$, are disabled. As illustrated in Fig. 9 the problem is again caused by the rendez-vous statements.

With $s$ we denote the state in the original state space in which process $C$ is about to execute the statement $x = 0$, the processes $A$ and $B$ are hanging on

\footnote{Note that if one tries to run the example with the recent releases of Spin an error will be issued because of the incompatibility of fairness and partial-order reduction in models with rendez-vous operations.}
Fig. 8. Motivating example for the fairness algorithm.

their rendez-vous sending statements, and the counter $C$ from the WF algorithm equals the Pid of process $B$. Then, the statement $x = 0$ is no longer safe, because it is dependent with the statement $\epsilon_{P_{id}(B)}$ (i.e. $\epsilon_C$). Namely, because $c!1$ is disabled, according to the WF algorithm the $\epsilon_{P_{id}(B)}$-statement is enabled. After the execution of $x = 0$ the system passes to the state $s'$ in which $c!1$ becomes enabled, and consequently the $\epsilon_{P_{id}(B)}$-statement is not possible anymore. On the other hand in $s'$, after $x = 0$, $c!1$ becomes enabled and must be included in the fair cycles. Since Spin assumes that $x = 0$ is safe, in the reduced search the verifier never considers the fair cycle in which process $B$ does not contribute a transition, which is, of course, wrong.

Solution to the Compatibility Problem. There is an apparent analogy with the pattern from the unless case. By enabling a rendez-vous statement (transition) ($c!1$ in combination with $c?x$) we are preventing another transition (in this case the $\epsilon_{P_{id}(B)}$ transition) which is in discord with the independence definition.

As in the case of the unless construct two kinds of solutions are possible. The first solution is static and it is actually the same with the one for the problem
with unless. This is not surprising because we have the same problematic reduction pattern which we can avoid exactly in the same way – by declaring as unsafe all statements that are safe according to the standard criteria in Spin, if they are followed by a rendez-vous receive statement.

A dynamic solution which is analogous to the one for the unless case, also looks plausible. In each state we need to check if there is a possibility of an $\epsilon$-transition move caused by a rendez-vous communication. The partial order reduction is not performed if this is the case. Unlike in the unless case, this time the time overhead can be much smaller because we have to check the transitions from only one process - the one who’s Pid equals the counter component $C$.

6 Conclusion

Promela’s unless construct and the weak fairness algorithm are both incompatible with the partial order reduction algorithm when rendez-vous communications are present in the programs. We gave solutions to both problems by proposing a corrected identification of safe statements or changes in the partial order algorithm. It is hoped that the lessons learned from these problems will be helpful to avoid the interference of the partial order with the prospective new features of Spin.

8 The author owes this observation to Dennis Dams.

9 After the publication of the original version of this chapter, we discovered that the same erroneous pattern can cause incorrect results when POR is used in models with priority choice (unless statements) and communications via so-called exclusive send and exclusive receive buffered channels. In some states the communication statements on such channels are also treated as (conditionally) safe (see [14]). One can construct a counter example which is very similar to the one in Fig. 2 which shows that more strict conditions for the safety of send and receive statements on exclusive channels are needed in presence of unless constructs. The solution to this problem proposed
A natural task for the future work would be the implementation of the solutions in Spin. As a first step towards the static solution we tested successfully a prototype implementation done by modifying the Spin mechanism for labeling the safe statements [3]. The first main problem was to find the successor of a given statement. The major difficulty in this context was the handling of the various Promela jump constructs (break, goto, etc.). The second more serious problem was the detection whether some channel is a synchronous (rendez-vous) one. We circumvented this by treating all the channels as rendez-vous, i.e., each statement which is followed by a receiving statement on any channel was treated as unsafe. Also the implementation of the first dynamic solution for the fairness should not be too involved.

As a final remark, the compatibility with the weak fairness algorithm can be important for the existing [17, 2] and future extensions of Spin with real time, especially having in mind the work of [4] about Zeno cycles in the real-time systems.

Acknowledgments The author is grateful to Dennis Dams, Gerard Holzmann, Victor Bos and Leszek Holenderski for their valuable information and comments.

References

7. D. Dams, private communication

by the author was implemented by Gerard Holzmann in the standard distribution of Spin (versions 3.3. and later).