Partial Order Reduction in Presence of Rendez-vous Communications with Unless Constructs 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 the Promela models, the `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 the incompatibility, we give solutions for 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 inherent and important one in the study of concurrency and nondeterminism, in particular in the area of the verification of concurrent systems. Since fairness is used as generic notion there is a broad taxonomy of fairness concepts. In this paper 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, 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 this (in the mutual exclusions) or eventually leaving the waiting queue for each process that has entered it (in the scheduling) [7].

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 [16, 8, 11, 14] 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 are interleaving 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, recently it was discovered 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 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 this three Spin's features is also currently forbidden.

Interestingly, it turns out that both problems are caused exactly by the same pattern of wrong partial order reduction. After pointing out the incorrect reduction pattern we propose solutions for these problems. For each of the problems we discuss two kind of solutions, classified according to the two different phases of the verification in which they are implemented. The first kind of solutions corrects the identifications of so called safe statements for the partial order reduction algorithm. The marking of the statements as safe is done during the compilation of the Promela model, so we call this 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 paper. 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 two solutions (one dynamic and one static) to overcome it. Section 5 deals with the Spin's weak fairness algorithm. After location of the problem and the comparison with the `unless` case, we again propose both kind of solutions. The last section is a standard summary with some considerations about the future work.

2 Preliminaries

In this section following [11] and [5] we give semantics of the 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 $(S_i, s_{0i}, \tau_i, L_i)$, where S_i is a finite set of states, s_{0i} is a distinguished initial state, L_i is a set of program statements (labels), and $\tau_i : S_i \times L_i \to 2^{S_i}$ is a nondeterministic transition function. The transition function induces a set $T_i \subseteq S_i \times S_i$ of transitions. Every transition in T_i is the result of an execution of a statement from the process, i.e., $(s_i, s'_i) \in T_i$ iff there exists a statement a such that $s'_i \in \tau_i(s_i, a)$. We introduce a function *Label* that maps each transition to the corresponding statement. For a statement a, with $Pid(a)$ we denote the process to which a belongs. (If two syntactically identical statements belong to different processes we consider them as different.) $En(a)$ denotes the set of states in which a is enabled. (The enabledness (executability) of a given statement is defined according to some additional rules that we do not consider here.) Given a state s and process p we say that p is enabled in s (or, more formally, write $s \in En(p)$) if there is a statement a such that $Pid(a) = p$ and $s \in En(a)$.

Now we can define the semantics of the program P that corresponds to the concurrent execution of the processes P_i as an LTS which is a product of the labeled transition systems corresponding to the component processes. The product LTS consists of: state space $S = \prod S_i$, i.e., the Cartesian product of the state spaces S_i from P_i's LTS; $s_0 = (s_{01}, \ldots, s_{0n})$; $L = \bigcup L_i$, the set of statements is union of the statements sets of the components; and the transition function is defined by: (i) if a is not a rendez-vous statement, then $(s_1, \ldots, s'_k, \ldots, s_n) \in \tau(s_1, \ldots, s_k, \ldots, s_n, a)$ iff $s'_k \in \tau_k(s_k, a)$; (ii)if a is a rendez-vous send, and there is a rendez-vous receive statement a', such that $Pid(a') \neq Pid(a)$, then $(s_1, \ldots, s'_k, \ldots, s'_l, \ldots, s_n) \in \tau(s_1, \ldots, s_k, \ldots, s_l, \ldots, s_n, a)$ iff $s'_k \in \tau_k(s_k, a)$ and $s'_l \in \tau_l(s_l, a')$. Note that in the case of rendez-vous communication the resulting transition is labeled with the rendez-vous send statement.

An execution sequence for the LTS is an infinite sequence $t_0, t_1, \ldots$ where $t_i \in T$, for all $i \geq 0$ and t_0 originates from the initial state, i.e. $t_0 = (s_0, s'_0)$ (for some s'_0) and for all adjacent transitions $t_{i+1} = (s_{i+1}, s'_{i+1}), t_i = (s_i, s'_i) (i \geq 0)$ it holds $s'_i = s_{i+1}$. The execution sequence can also be defined as a sequence of global system states. In order to relate the two options for a given transition $t_i = (s_i, s'_i)$ we define $State(t_i) = s_i$.

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

A *Büchi automaton* is a tuple $A = (\Sigma, S, \rho, s_0, F)$, where Σ is an alphabet, S is a set of states, $\rho : S \times \Sigma \rightarrow 2^S$ is a nondeterministic transition function, $s_0 \in S$ is the initial state, and $F \subseteq S$ is a set of designated states. A *run* of A over an infinite word $w = a_1 a_2 \ldots$, is an infinite sequence $s_0 s_1 \ldots$ of states, where s_0 is the initial state and $s_i \in \rho(s_{i-1}, a_i)$, for all $i \geq 1$. A run $s_0 s_1 \ldots$ is *accepting* if there is some state from F that occurs infinitely often, i.e. for some $s \in F$ there are infinitely many i's such that $s_i = s$. The word w is *accepted* by A if there is an accepting run of A 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.

In order to prove the satisfaction of the LTL formula by the program, we further define the *synchronous* product of LTS (S_P, s_{0P}, τ, L) corresponding to the program P and the Büchi automaton $(\Sigma, S_A, \rho, s_{0A}, F)$ obtained from the negation of the LTL formula, to be an LTS extended with acceptance states, [1] with: state set $S_P \times S_A$, initial state (s_{0P}, s_{0A}), transition function $\tau : S_P \times$

[1] Although the proliferation of different formal models (LTS, Büchi automata, ex-

$S_A \times L$ 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_{1A}, set of statements L, and set of designated acceptance states Acc defined such that $(s_P, s_A) \in Acc$ iff $s_A \in F$, i.e. we declare as acceptance states the states with second component belonging to the acceptance set of the Büchi automaton. Similarly as for Büchi automata we will say for an execution σ that it is acceptance execution if there is at least one state from Acc that occurs infinitely often in σ.

The satisfaction of the formula can now be proven by showing that there are no acceptance executions of the extended LTS. On the other hand, the existence of acceptance executions sequences means that the formula is not satisfied. From the definition of Büchi automata and extended LTS and following the reasoning from [5], for instance, it is straightforward to conclude that the extended LTS has an acceptance execution iff it has some state $f \in Acc$ that is reachable from the initial state and reachable from itself (in one or more steps) [5]. In the sequel we will call the underlying graph a *state space*. Thus, we have to look for *acceptance cycles* in the state space, i.e., for cycles that contain at least one acceptance state.

3 Partial Order Reduction

In this section we give a brief overview of the partial order reduction (POR) algorithm by Holzmann and Peled [11], that is considered throughout the paper. This algorithm is also implemented in Spin. We start with rephrasing some definitions from [11].

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 behaviour, then $vbaw$ is an accepted behaviour 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. For $q \in En(a)$, $a(q)$ is state which is reached by executing a in state q. Concurrent statements (i.e. statements with different 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

tended LTS) that are used to represent the semantics and ultimately the state space might seem unnecessary, we use three different formal concepts in order to follow more closely [11] and [5], so that we will be able to reuse most of the results from these papers in a seamless way.

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. [16, 8]). The advantage of the subtlety in Definition 1 is that 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 paper.

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 the section we describe in a rather informal way the partial order algorithm from [11]. For the full details about the algorithm we recommend the original references [11, 14].

The reduction of the search space is effected during the DFS, by limiting the search from a state s to a subset of the statements 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 which has only safe statements enabled and all those transitions lead to a state which is not on the DFS stack, then the ample set consists of all the statements from this process only. Otherwise, the ample set consists of all enabled statements in s. It can be proven [11, 14] that the reduced graph obtained in this way preserves the properties of the original LTS, stated as an LTL formula. 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 it 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. For instance, 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 correct (or better to say, refine) this safety criterion.

In [11, 14] it is shown that

Theorem 2. *If there exist reachable acceptance cycles in the state space the reduced search algorithm will report at least one of them.*

4 The `unless` construct

The `unless` construct is a mean for modeling exception handling routines and priority choices. Formally, it introduces partial ordering between the statements that belong to a same process. Its syntax is *stmnt* `unless` *stmnt*. The first (left-hand) statement is called *normal* or *main*, while the second (right-hand) is *escape* statement. [2] 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 the problems when the partial order reduction is used and the escape statement is a rendez-vous communication.

Let us consider the motivating Promela example given in Figure 1. [3] (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 only 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)`. [4] However, in the reduced search this is not detected, because of the incorrect partial order

[2] In general, both statements can be sequences of Promela statements. Also the `unless` construct can be nested. The results form this paper can be extended in a straightforward way for this general case.

[3] 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 draw our attention to the possible problems with the `unless` statement.

[4] 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 decided to use this version of the example for the sake of simplicity.

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

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


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

Fig. 1. Motivating example for unless statement.

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 of the specific property that the executability of the rendez-vous statement can be changed only because of the change of the location in the process (program counter), the statements like `skip` are not unconditionally globally independent according to the Definition 1.

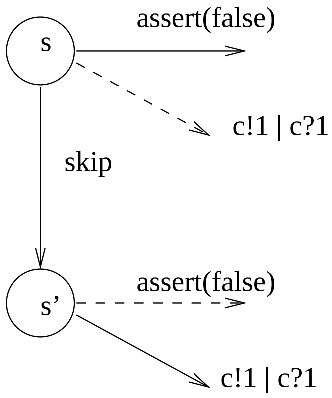

Fig. 2. Interdependence between the "safe" statements and the `unless` escape and normal statements.

The reason is depicted in Fig. 2. In the starting state described above the rendez-vous send `c!1` is disabled, but with the execution of `skip` it becomes enabled. This means that `skip` has indirectly disabled `assert(false)` which was enabled in the starting state. In that way `skip` and `assert(false)` are not independent according to the Definition 1, because the net effect is that `skip` disables `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 a 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 by a rendez-vous communication (more precisely, by a rendez-vous receive) as unsafe. We use the term "followed" in a syntactical sense, taking into account all the cycles and jumps in the program. 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 DFS there is no disabled rendez-vous send in `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 the things can become complicated because of the feature interaction.

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

The weak fairness algorithm that we are going to present in the next subsection is an extension of the algorithm of Courcoubetis, Vardi, Wolper and Yannakakis [5] 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 product LTS. It is also compatible with Spin's bit-state hashing, which is not the case with the strongly connected components algorithm. We start with the brief overview of this algorithm given in Figure 3.

The algorithm consists of two alternating depth first searches for exploration of the state space. When the first search retracts to an acceptance state the second search is started to check for a cycle through this acceptance state. If the second depth first search (DFS) closes a cycle, it is reported as an acceptance cycle. Otherwise, the first DFS resumes until the next acceptance state in postorder (if there is such a state). If no cycle is found than 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 a different copy we extend the state with one bit denoting the DFS phase.

```
proc dfs(s)
   add {s,0} to Statespace
   for each successor s' of s  do
       if {s',0} not in Statespace then dfs(s') fi
   od
   if accepting(s) then seed:={s,1}; ndfs(s) fi
end

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

Fig. 3. Nested depth first search algorithm.

The following theorem from [5] establishes the correctness of the algorithm

Theorem 3. *The algorithm in Fig. 3 reports a cycle iff there is an acceptance state reachable from the initial state that belongs to a nontrivial strongly connected component of the state space.*

Notice that the first DFS serves to order the acceptance states in postorder, such that the cycle checks can be done starting from the acceptance state which is removed first from the stack (i.e. first in postorder). It is important to emphasize that during the second DFS each state in the second copy of the state space is visited only once. This has (most of the time) an advantage over the straightforward solution to do the cycle check in preorder, i.e. starting a cycle check as soon as we visit an acceptance state. If we used preorder we would have to start the second DFS always from scratch. Although the memory requirements would be the same as for the postorder, the multiple visits to a same state can lead to a significant time overhead.

5.2 Description of the Weak Fairness Algorithm

We will consider weak fairness with regard to processes, i.e. we will 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 4. An execution sequence $\sigma = t_0 t_1 \dots$ is fair iff for each process p the following holds: for all $i \geq 0$ such that $State(t_j) \in En(p)$ for all $j \geq i$, there is $k \geq i$ such that $Pid(Label(t_k)) = p$.

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 basic idea behind the weak fairness algorithm is to work in an extended state space instead of the original one. The extended state space consists of N copies of the original state space, where N is the number of processes. Whenever we are in the i-th copy and either we take a transition belonging to the process i or there is no executable transition that belongs to i, then we pass to the $((i+1) \mathrm{m}od N) + 1$ copy. A cycle is then fair if and only if it passes through all the copies. This is because we are sure that all permanently enabled processes have contributed an action to the cycle.

The idea is quite straightforward and at first sight it seems that all we need is some counter which will indicate the copy of the state space we are passing through. However, the algorithm becomes more involved when we have to take care about the partial order reduction and try to improve it with some heuristics.

The weak fairness (WF) algorithm we are considering here is by Holzmann and it is a variant of the Choueka's flag algorithm [10]. This algorithm is also implemented in Spin as an extension of the nested depth first search algorithm.

For the WF algorithm we need a new extended state space. Its elements are quadruples of the form (s, A, C, b), obtained by extending each state s, apart from the bit b discriminating between the first and second DFS, with an additional bit A and an integer C. The variable C is the already mentioned counter that keeps track of the copy of the state space we are passing through. The role of A is to indicate that we have already passed through an acceptance state in the state space. We need this indicator because, as we will see in a moment, we will not always start the search from an acceptance state, but instead in a state when we are sure that all enabled processes have executed a statement, which often means that we have a better chance to close a fair cycle.

The standard NDFS model-checking algorithm is modified correspondingly for manipulation of the new variables and generation of the extended state space. The pseudo-code of the algorithm is given in Figure 4 and 5.

In the initial state A and C are zero. The values of A and C are changed according to the following three rules that apply both to the first and the second DFS.

- Rule 1: If A is zero in an acceptance state, then A is set to 1 and C is assigned the value $N + 1$, where N is the number of processes. (This rule is a kind of initialization that indicates that we have passed an acceptance state and a cycle check, i.e., a second DFS, is possibly needed.)
- Rule 2: If A is 1 and C equals the Pid number (increased by two) of the process which is being currently considered, then C is decreased by 1. (This rule is to keep track of the state space copy in which we are working. The technical difference is that the counter is increased instead of being decreased, as in the "naive" algorithm above.)
- Rule 3: If the condition of Rule 1 does not apply and if C is 1, then both C and A are reset to 0. (This rule is a preparation (initialization) for the second DFS.)

We have to use the counter not only in the second DFS, but also during the first DFS in order to keep the copies of the state space the same in both searches. This is important for the partial order reduction algorithm that we are going to use later. If those copies differ, the interaction between the nested DFS algorithm and the partial order reduction can produce incorrect results, as shown in [12].

Notice that if C is $i + 2$, then it is immediately decreased when we start exploring the statements (transitions) of a new process. This means that even if the process cannot perform any statement we will pass to a new state with a different counter number. This kind of transition in the state space we call a *default transition* and label it with a *default statement* meaning that the process is disabled and should not be taken into consideration for the cycle that is currently checked. We will see in the next subsection that these default transitions play a crucial role in the incompatibility with partial order reduction.

We mentioned above that unlike the standard NDFS in the WF algorithm the cycle check does not have to start in an acceptance state. Instead it starts in special states, which we simply call *starting states* in the sequel, and which have the property that $A = 1$, $C = 1$ and $b = 1$. Analogously to the standard NDFS, the cycle check starts after all the successors of the starting state are explored and the search returns back to the starting state. Also, let us emphasize that a starting state can "cover" several acceptance states in a sense that several acceptance states can be passed before the cycle check is started. The additional acceptance state that are passed after the first one do not "generate" new starting states because C stays unchanged. In this way we can often reduce the number of calls for the second DFS. Also by the lookahead for a possible fragment of a fair cycle, as we essentially do, we reduce the number of unsuccessful attepmpts to close a fair cycle.

5.3 Incompatibility with the Partial Order Reduction Algorithm

The incompatibility of the weak fairness algorithm with POR (and because of rendez-vous communications) was first discovered for the example given in Fig. 6 (by Dennis Dams):

```
proc dfs(s, A, C)
   add {s,A,C,0} to Statespace
   /* Rule 1 */
   if accepting(s) then
      if A == 0 then
         A = 1;
         C = N+1
      fi
   else
   /* Rule 3 */
      if C == 1 then
         A = 0;
         C = 0
      fi
   for each process i = N-1 downto 0 do
       /* Rule 2 */
       if A == 1 and C == i+2 then
          C = C - 1
       fi
       nxt = all transitions enabled in s with Pid(t)=i
       for all t in nxt do
          s' = successor of s via t
          if {s',A,C,0} not in Statespace then dfs(s',A,C) fi
       od
   od
   if A == 1 and C == 1 then
      seed:={s,A,C,1};
      ndfs(s,A,C)
   fi
end
```

Fig. 4. Weak fairness algorithm – first depth first search (continued in Figure 5).

In the model from Fig. 6 the LTL formula $\Diamond p$, where p is defined as $b == true$, is not valid even under fairness condition. This is because of 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 ...` along which the process `B` can be safely ingnored because it is not continuously enabled.

However, when partial order redcution was used the verifier uncorrectly showed that the forumula was valid. [5] So, the aforementioned fair cycle formed

[5] Note that if you try 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.

```
proc ndfs(s,A,C) /* the nested search */
   add {s,A,C,1} to Statespace
   /* Rule 1 */
   if accepting(s) then
      if A == 0 then
         A = 1;
         C = N+1
      fi
   else
   /* Rule 3 */
      if C == 1 then
         A = 0;
         C = 0;
      fi
   for each process i do
       /* Rule 2 */
       if A == 1 and C == i+2 then
          C = C - 1
       fi
       nxt = all transitions enabled in s with Pid(t)=i
       for all t in nxt do
          s' = successor of s via t
          if {s',A,C,1} not in Statespace then ndfs(s',A,C) fi
          else if {s',A,C,1}==seed then report cycle fi
       od
   od
end
```

Fig. 5. Weak Fairness Algorithm – second depth first search

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 of a wrong partial order reduction because of the incorrect independence relation occurs. But this time the problem is with the difault meta transitions. Recall that those transitions only change the counter C in the fairness algorithm when all statements of the process which is currently considered are disabled. As ilustrated in Fig. 7 the problem is again caused by the rendez-vous statements.

With s is denoted the state in which process C is about to execute the statement `x = 0`, processes A and B are hanging on their rendez-vous sending statements, and the counter C from the WF algorithm equals the Pid of process B increased by 2. Then, the statement `x = 0` is no longer safe, because it is dependent with the default transition (the decrement of C). Namely, because `c!1` is disabled, according to the WF algorithm the default transition is enabled. After the execution of `x = 0` the system passes in the state s' in which `c!1` becomes enabled, and consequently the default transition is not possible. In the reduced search the statement `c!1` is not considered at all. On the other

```
chan c =[0] of {bit};
bool b = false;

active proctype A()
{
starta:
  c!0;
  goto starta
}

active proctype B()
{
startb:
  c!1; b = true;
  goto startb
}

active proctype C()
{ bit x;
startc:
  c?x; x = 0;
  goto startc
}
```

Fig. 6. Motivating example for the fairness algorithm.

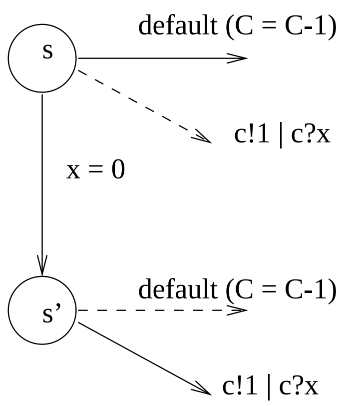

Fig. 7. Interdependence between the "safe" transitions and the default moves

hand in s', after `x = 0`, `c!1` becomes enabled and must be included in the fair cycles. In this way during the reduced search the verifier never considers the fair cycle in which process B does not contribute a transition, which is, of course, wrong. 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 "meta" default transition) which is in discord with the independence definition.

As in the case of the `unless` construct two kind of solutions are possible.

The first solution is static and it is actually the same with the one for the problem with `unless`. [6] 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.

We can also propose a dynamic solution which is analogous to the one for the `unless` case. In each state we need to check if there is a possibility of a default 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 satisfies $C = Pid + 2$.

However, we conjecture that there is a much more efficient dynamic solution that still allows reduction even in the state with a default move. In the new solution we use again the fact that we exactly know the possible problematic statements, or more precisely the Pid of the process which default move can cause the incorrect reduction.

In fact we correct the POR algorithm, while the same static criteria for safety of statements remain unchanged as in the standard POR. The change of the algorithm follows from the following reasoning. Because the problem is with the default move we should take care that we do not loose the counter decrement. To this end, before doing a reduction on a safe transition we first check if the process with Pid such that $C = Pid+2$ can make a default move because no one of its transitions is enabled in the current state and in the same time contains a synchronous communication (which is not enabled, of course). If there is such a process we execute this default move before the reduction. In that way we do not loose the counter decrement because of the incorrect reduction. By decreasing C now another process can acquire the above problematic property to be blocked and to have a rendez-vous send while his Pid equals $C-2$. We must obviously do the check described above in an iteration, until there are no processes with the problematic property. Although this solution will also exhibit a time overhead during the verification because of the necessary checks, it is reasonable to expect that this overhead will not be that significant. The new dynamic solution keeps of course the advantage over the static one that it can provide better reduction of the state space. Probably only the prospective practical implementations and tests on case examples can answer the question whether the possible timing overhead will be compensated by significantly less memory consumption. The

[6] The author owes this observation to Dennis Dams.

disadvantage for the practical implementation can be that the second dynamic solution (at least conceptually) seems more complicated than the static one.

As we stated above, our proposal is only a conjecture for the time being, because we are still checking the proofs. The idea is to show by an adaptation of the correctness proof from [11, 14] that with the small addendum described above the POR algorithm remains correct even when the reduction is done on in fact unsafe statements (with regard to the default move). This is because the LTL formulae do not refer to C which is a "meta" variable in the system and affects only the cycle fairness.

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.

A natural task for the future work would be the implementation of the solutions in Spin. In that regard the most promising looks the static solution with the correction of the criteria for safe statements. The only obstacle can be to find the successor of a given statement. The main problem in this context is the handling of the various Promela jump constructs (`break`, `goto`, etc.). Also the implementation of the first dynamic solution for the fairness should not be too involved. In the theoretical direction the correctness proof of the conjectured dynamic solution for fairness remains to be rechecked.

As a final remark, the compatibility with the weak fairness algorithm can be very important for the existing [2] and future extensions of Spin with real time, especially having in mind the work of [3] 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

1. Bengtsson, J., Jonsson, B, Lilius, J., Yi, W., *Partial Order Reductions for Timed Systems*, CONCUR'98, LNCS 1466, pp. 485-501, 1998.
2. Bošnački, D., Dams, D., *Integrating Real Time in Spin: a Prototype Implementation*, FORTE/PSTV'98, Kluwer, pp. 423-439, 1998
3. Bouajjani, A., Tripakis, S., Yovine, S., *On-the-Fly Symbolic Model-Checking for Real-Time Systems*, In Proc. of the 18th IEEE Real-Time Systems Symposium, pp. 232-243, IEEE, 1997

4. Clarke, E., Emerson, E., Sistla, A., *Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications*, ACM Transactions on Programming Languages and Systems, 8(2), pp. 244-263, 1986
5. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M., *Memory Efficient Algorithms for the Verification of Temporal Properties*, Formal Methods in System Design I, pp. 275-288, 1992
6. Dams, D., Gerth, R., Knaack, B., Kuiper, R., *Partial-order Reduction Techniques for Real-time Model Checking*, FMICS'98, CWI, pp.157-170, 1998
7. Francez, N., *Fairness*, Springer, 1986
8. Godefroid, P., *Partial Order Methods for the Verification of Concurrents Systems: An Approach to the State Space Explosion*, LNCS 1032, Springer, 1996
9. Holzmann, G. J., *Design and Validation of Communication Protocols*, Prentice Hall, 1991. Also: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`
10. Holzmann, G. J., Personal communication
11. Holzmann, G., Peled, D., *An Improvement in Formal Verification*, FORTE 1994, Bern, Switzerland, 1994.
12. Holzmann, G., Peled, D., Yannakakis *On Nested Depth First Search*, Proc. of the 2nd Spin Workshop, Rutgers University, New Jersay, USA, 1996.
13. Pagani, F., *Partial Orders and Verification of Real Time Systems*, Formal Techniques in Real Time and Fault Tolerant Systems FTRTFT 96, LNCS, 1996
14. Peled, D., *Combining Partial Order Reductions with On-the-Fly Model Checking*, Computer Aided Verification 1994, LCNS 818, pp. 377-390, 1994.
15. Vardi, M., Wolper, P., *Automata Theoretic Techniques for Modal Logics of Programs*, Journal of Computer and System Science, 32(2), pp. 182-221, 1986
16. Willems, B., Wolper, P., *Partial Order Models for Model Checking: From Linear to Branching Time*, Proc. of 11 Symposium of Logics in Computer Science, LICS 96, New Brunswick, pp. 294-303, 1996