Static Reductions for Promela Specifications

C. Pajault and J.-F. Pradat-Peyre

CEDRIC - CNAM Paris
292, rue St Martin, 75003 Paris
{pajault,peyre}@cnam.fr

Abstract. The interleaving of concurrent processes actions leads to a combinatory explosion. There exists in Petri nets theory some structural reductions that combat the state explosion by agglomerating sequences of transitions into a single atomic transition. These reductions are easily checkable and preserve deadlocks, Petri nets liveness and any LTL formula that do not observe the modified transitions. Furthermore, they can be combined with others kinds of reductions such like partial-order techniques to obtain very effective reductions. We propose in this paper to adapt these reductions to Promela specifications by proposing some simple rules which give the possibility to automatically infer atomic steps in the Promela model while preserving the checked property. We demonstrate on typical example the efficiency of this approach and we propose some perspectives of this work.

1 Introduction

The interleaving of concurrent processes actions leads to a combinatory explosion. In order to give a simple insight about this problem, let us take a small example. Let $\{p_i\}_{i=1\ldots n}$ be a set of stateless servers which infinitely execute a loop consisting in a sequence of two actions $accept_i$ and $execute_i$. The interleaving of these actions leads to a state space whose size is 2^n. Partial order methods (e.g. persistent sets [WG93], sleep sets [GW93], stubborn sets [Val93], ...), or symmetry based reductions [EPS93,AHI98,Sis03] may reduce the size of the state space to a size of n. However, the simple fact of considering the sequence as atomic leads to a state space reduced to a singleton! Obviously, as for partial order techniques, such a reduction may be faulty since for instance, it could hide occurrence of deadlocks. Thus the goal of a reduction theory is to (syntactically) characterize situations where a reduction is sound and how to perform it.

We proposed in [HPP04] new Petri nets reductions based on this principle and that cover a large range of patterns by introducing algebraic conditions whereas the previously defined ones [Ber83,Ber85] rely solely on structural conditions. We extended in [EHPP04b,EHPP04a] these "ordinary" reductions to colored Petri nets which are an abbreviation of Petri nets (to each colored net corresponds an ordinary Petri net). Indeed, colored Petri nets are a concise formalism for the modeling of concurrent software and they can be automatically derived from software using a tool such as Quasar [EKPPR03] which aims to verify concurrent Ada programs. So, reducing colored Petri nets is an efficient way to simplify concurrent software analysis.

We propose here to study how these new reductions can be used in Promela specifications analysis. First we show that we can translate a Promela model into a colored Petri net using a methodology already applied in the Quasar tool. Then we show that the verification of this colored Petri net can be efficiently performed using both structural agglomeration reductions and stubborn sets techniques. In order to avoid translation of a Promela specification into a colored Petri nets model we propose some simple syntactical rules based on Petri nets agglomerations which allow us to automatically detect sequences of statement that can be marked as "atomic" (using the `atomic` construction of Promela) while preserving analyzed properties. We show on classical examples that by this way we can significantly reduced the size of the state space without increasing time analysis.

2 Petri nets transitions agglomerations

A Petri net reduction is characterised by some application conditions, by a net transformation and by a set of preserved properties (i.e. which properties are simultaneously true or false in the original net and in the reduced one). Before presenting the "behavioral" version of the pre- and the post-agglomerations we first recall some Petri nets definitions.

2.1 Brief Petri nets definitions and notations

Definition 1. *A marked net* (N, m_0) *is a tuple* (P, T, W^-, W^+, m_0) *where:*

- P *is the finite set of places,*
- T *is the finite set of transitions disjoint from* P*,*
- W^- *(resp.* W^+*) an integer matrix indexed by* $P \times T$ *is the backward (resp. forward) incidence matrix,*
- m_0 *a integer vector indexed by* P *is the initial marking.*

Notations

- We note $\langle P, T, W^+, W^-, m_0 \rangle$ a marked Petri net;
- λ defines the empty sequence of transitions;
- If s is a sequence of transitions, $|s|$ denotes the length of s (that is recursively defined by $|\lambda| = 0$ and $|s.t| = |s| + 1$);
- $\Pi_{T'}(s)$ denotes the projection of the sequence s on a subset of transitions T' and is recursively defined by $\Pi_{T'}(\lambda) = \lambda$, $\forall t \in T'$, $\Pi_{T'}(s.t) = \Pi_{T'}(s).t$ and $\forall t \notin T'$, $\Pi_{T'}(s.t) = \Pi_{T'}(s)$,
- $|s|_{T'} = |\Pi_{T'}(s)|$ denotes the number of occurrences of transitions of T' in s.
- $Pref(s) = \{s' \mid \exists s'' \text{ s.t. } s = s'.s''\}$ denotes the set of prefixes of s.

Definition 2. *Let* (N, m_0) *be a marked net then:*

- $t \in T$ *is firable from* m *a marking (denoted* $m[t\rangle$*) iff* $\forall p \in P$ $m(p) \geq W^-(p, t)$*,*

- *the firing of $t \in T$ firable from m leads to the marking m' (denoted $m[t\rangle m'$) defined by $\forall p \in P$ $m'(p) = m(p) + W(p,t)$ where W the incidence matrix is defined by $W = W^+ - W^-$. Note that the incidence matrices W, W^- and W^+ can easily be extended to matrices indexed by $P \times T^*$.*

Definition 3. *Let (N, m_0) be a marked net then:*

- *$s \in T^*$ is firable from m a marking and leads to m' (also denoted by $m[s\rangle$ and $m[s\rangle m'$) iff*
  1. *either $s = \lambda$ and $m' = m$*
  2. *or $s = s_1.t$ with $t \in T$ and $\exists m_1$ $m[s_1\rangle m_1$ and $m_1[t\rangle m'$*
- *$s \in T^\infty$ is firable from m a marking (also denoted $m[s\rangle$) iff for every finite prefix s_1 of s, $m[s_1\rangle$.*

Definition 4. *Let (N, m_0) be a marked net then:*

- *$Reach(N, m_0) = \{m | \exists s \in T^*\ m_0[s\rangle m\}$ is the set of reachable markings,*
- *m is a dead marking if $\forall t \in T$ $NOT(m[t\rangle$,*
- *(N, m_0) is live iff $\forall m \in Reach(N, m_0)\, \forall t \in T\, \exists s \in T^*\ m[s.t\rangle$,*
- *$L(N, m_0) = \{s \in T^* | m_0[s\rangle\}$ is the language of finite sequences,*
- *$L^{Max}(N, m_0) = \{s \in T^* | \exists m\, dead\, marking\, m_0[s\rangle m\}$ is the language of finite maximal sequences,*
- *$L^\infty(N, m_0) = \{s \in T^\infty | m_0[s\rangle\}$ is the language of infinite sequences,*

2.2 Petri nets agglomerations

We note (N, m_0) a Petri net and we suppose in the following definitions that the set of transitions of the net is partitioned as: $T = T_0 \biguplus_{i \in I} H_i \biguplus_{i \in I} F_i$ where I denotes a non empty set of indices. The underlying idea of this decomposition is that a couple (H_i, F_i) defines transitions sets that are causally dependent: an occurrence of $f \in F_i$ in a firing sequence may always be related to a previous occurrence of some $h \in H_i$ in this sequence. Starting from this property, we developed conditions on the behaviour of the net which ensure that we can restrict the dynamics of the model to sequences where each occurrence $h \in H_i$ is immediately followed by an occurrence of some $f \in F_i$ without changing its behaviour w.r.t. to a set of properties. This restricted behaviour is the behaviour of a reduced net as shown in the next definitions and propositions.

Definition 5 (Reduced net). *The reduced Petri net (N_r, m_{0r}) is defined by:*

- *$P_r = P$, $T_r = T_0 \cup_{i \in I} (H_i \times F_i)$ (we note hf the transition (h, f) of $H_i \times F_i$);*
- *$\forall t_r \in T_0, \forall p \in P_r$, $W_r^-(p,t) = W^-(p,t)$ and $W_r^+(p,t) = W^+(p,t)$*
- *$\forall i \in I, \forall hf \in H_i \times F_i, \forall p \in P_r$ $W_r^-(p, hf) = W^-(p, h.f)$ and $W_r^+(p, hf) = W^+(p, h.f)$*
- *$m_{0r} = m_0$*

From now, we note $H = \cup_{i \in I} H_i$ and $F = \cup_{i \in I} F_i$. The firing rule in the reduced net is noted $\rangle_r$ (i.e. $m[s\rangle_r m'$ denotes a firing sequence in the reduced net). We note also ϕ the homomorphism from the monoid T_r^* to the monoid T^* defined by:

$$\forall t \in T_0, \phi(t) = t \text{ and } \forall i \in I, \forall h \in H_i, \forall f \in F_i, \phi(hf) = h.f$$

This homomorphism is extended to an homomorphism from $\mathcal{P}(T_r^*)$ to $\mathcal{P}(T^*)$ and from $\mathcal{P}(T_r^\infty)$ to $\mathcal{P}(T^\infty)$.

The next basic proposition states in a formal way that the behaviour of the reduced net is a subset of the original behaviour.

Proposition 1. *Let (N, m_0) be a net. Then:*

1. $\forall s_r \in T_r^*, m[s_r\rangle_r m' \iff m[\phi(s_r)\rangle m'$
2. $\forall s_r \in T_r^\infty, m[s_r\rangle_r \iff m[\phi(s_r)\rangle$

In order to obtain the preservation of more properties (such like deadlock occurences) we have to introduce new behavioural hypotheseses. The basic one, named *Potential agglomerability* ensures that an occurence of a transition of F is always preceeded by an occurence of a transition of H. For doing that we define a set of counting functions, denoted Γ_i, by $\forall s \in T^*$, $\Gamma_i(s) = |s|_{H_i} - |s|_{F_i}$.

Definition 6 (Potentially agglomerability). *A marked net (N, m_0) is potentially agglomerable (p-agglomerable for short) iff $\forall s \in L(N, m_0)$, $\forall i \in I$, $\Gamma_i(s) \geq 0$.*

We define now the behavioral conditions that ensure that the agglomerations preserve properties of the net. Note that these behavioral conditions can be checked with efficient structural and algebraical sufficient conditions (not presented here).

Pre-Agglomeration The following definition states four conditions which “roughly speaking” ensure that delaying the firing of a transition $h \in H_i$ until some $f \in F_i$ fires does not modify the behaviour of the net w.r.t. the set of properties we want to preserve.

Definition 7. *Let (N, m_0) be a p-agglomerable net. (N, m_0) is*

1. ***H-independent*** *iff $\forall i \in I$, $\forall h \in H_i$, $\forall m \in Reach(N, m_0)$, $\forall s$ such that $\forall s' \in Pref(s)$, $\Gamma_i(s') \geq 0$, $m[h.s\rangle \Longrightarrow m[s.h\rangle$*
2. ***divergent-free*** *iff $\forall s \in L^\infty(N, m_0)$, $|s|_{T_0 \cup F} = \infty$*
3. ***quasi-persistent*** *iff $\forall i \in I, \forall m \in Reach(N, m_0)$, $\forall h \in H_i$,*
   $\forall s \in (T_0 \cup F)^*$, such that $m[h\rangle$ and $m[s\rangle$ $\exists s' \in (T_0 \cup F)^*$ fulfilling: $m[h.s'\rangle$, $\Pi_F(s') = \Pi_F(s)$ and $W(s') \geq W(s)$.
   Furthermore, if $s \neq \lambda \Longrightarrow s' \neq \lambda$ then the net is **strongly** *quasi-persistent.*
4. ***H-similar*** *iff $|I| = 1$ or $\forall i, j \in I, \forall m \in Reach(N, m_0)$, $\forall s \in T_0^*$,*
   $\forall h_i \in H_i$, $\forall h_j \in H_j, \forall f_j \in F_j$ $m[h_i\rangle$ and $m[s.h_j.f_j\rangle \Longrightarrow \exists s' \in (T_0)^*$, $\exists f_i \in F_i$ such that $m[s'.h_i.f_i\rangle$ and such that $s = \lambda \Longrightarrow s' = \lambda$.

The H-independence roughly means that once a transition $h \in H_i$ is fireable it can be delayed as long as one does not need its occurrence to fire a transition of F_i. When a net is divergent-free it does not generate infinite sequences with some suffix included in H. In the pre-agglomeration scheme, we transform original sequences by permutation and deletion of transitions to simulateable sequences. Such an infinite sequence cannot be transformed by this way into an infinite simulateable sequence. Therefore this condition is mandatory. The quasi-persistence ensures that in the original net a "quick" firing of a transition of H does not lead to some deadlock which could have been avoided by delaying this firing. At last, the H-similarity forbids situations where the firing of transitions of F is prevented due to a "bad" choice of a subset H_i.

Under previous conditions (or a subset of), fundamental properties of a net are preserved by the pre-agglomeration reduction. This result is stated in the following theorem whose demonstration is provided in [HPP04].

Theorem 1. *Let (N, m_0) be a Petri net.*

1. *If (N, m_0) is p-agglomerable, H-independent and divergent-free then*

$$\Pi_{T_0 \cup F}(L^{max}(N, m_0)) \supseteq \Pi_{T_0 \cup F}(\Phi(L^{max}(N_r, m_{0r})))$$

2. *If (N, m_0) is p-agglomerable, H-independent, strongly quasi-persistent and H-similar then*

$$\Pi_{T_0 \cup F}(L^{max}(N, m_0)) \subseteq \Pi_{T_0 \cup F}(\Phi(L^{max}(N_r, m_{0r})))$$

3. *If (N, m_0) is p-agglomerable and H-independent then*

$$\Pi_{T_0 \cup F}(\phi(L^{\infty}(N_r, m_0))) = \Pi_{T_0 \cup F}(L^{\infty}(N, m_0))$$

The first point defines which conditions ensure that the reduction does not introduce maximal blocking sequences (e.g. characterizing a deadlock) in the reduced net. The second one fixes when the reduction does not hide some maximal blocking sequences. At last, the third point focuses on the preservation of properties expressed with infinite sequences (e.g. fairness properties).

Post-Agglomeration The main behavioural property that the conditions of the post-agglomeration implies is the following one: in every firing sequence with an occurrence of a transition h of H followed later by an occurrence of a transition f of F, one can immediately fire f after h. From a modelling point of view, the set F represents local actions while the set H corresponds to global actions possibly involving synchronisation.

Definition 8. *Let (N, m_0) be a p-agglomerable marked net. (N, m_0) is*

1. ***F-independent*** *iff $\forall i \in I$, $\forall h \in H_i$, $\forall f \in F_i$, $\forall s \in (T_0 \cup H)^*$, $\forall m \in Reach(N, m_0)$, $m[h.s.f\rangle \implies m[h.f.s\rangle$*
   (N, m_0) is ***strongly F-independent*** *iff $\forall i \in I$, $\forall h \in H_i$, $\forall f \in F_i$, $\forall s \in T^*$ s.t. $\forall s' \in Pref(s), \Gamma(s') \geq 0$ $\forall m \in Reach(N, m_0)$, $m[h.s.f\rangle \implies m[h.f.s\rangle$*

2. *F-**continuable** iff $\forall i \in I$, $\forall h \in H_i$, $\forall s \in T^*$, s.t. $\forall s' \in Pref(s), \Gamma(s') \geq 0$ $\forall m \in Reach(N, m_0)$ $m[h.s\rangle \Longrightarrow \exists f \in F_i$ such that $m[h.s.f\rangle$*

We express the strong dependence of the set F on the set H with these two hypotheses. The F-independence means that any firing of $f \in F$ may be anticipated just after the occurrence of a transition $h \in H$ which "makes possible" this firing. The F-continuation means that an excess of occurrences of $h \in H$ can always be reduced by subsequent firings of transitions of F.

As for the pre-agglomeration, these conditions (or a subset of) ensure that fundamental properties of a net are preserved by the post-agglomeration reduction (the demonstration is provided in [HPP04]).

Theorem 2. *Let (N, m_0) be a Petri net.*

1. *If (N, m_0) is p-agglomerable, F-continuable and F-independent then*

$$\Pi_{T_0 \cup H}(L^{max}(N, m_0)) = \Pi_{T_0 \cup H}(\Phi(L^{max}(N_r, m_{0r})))$$

2. *If (N, m_0) is p-agglomerable, F-continuable and strongly F-independent then*

$$\Pi_{T_0 \cup H}(\phi(L^{\infty}(N_r, m_0))) = \Pi_{T_0 \cup H}(L^{\infty}(N, m_0))$$

3 Using agglomeration for Promela model verification

We propose in this section to study how Petri nets agglomerations can be used to simplify the verification of Promela models. The first way consists in translating Promela specification into colored Petri nets before reducing and analyzing it. This methodologie gives good results in term of state space reduction but need the implementation of a complete translator and to deal with many Promela subtle statements.

So we propose a second approach, very easily applicable and which give almost as good result than the previous one. This method consists in syntactically detecting possible agglomeration in a Promela model (using **implicitly** the corresponding colored Petri net pattern) and in inferring automatically atomic sequences with the *atomic* Prolema construction.

3.1 Promela specifications and colored Petri nets

Background We presented in [EKPPR03] a tool named QUASAR which aims to automatically analyses concurrent Ada programs. This tool is based on colored Petri nets and translate concurrent Ada programs into colored Petri nets with the help of pre-defined patterns : each statement of the program is translated into a sub-nets following the corresponding pattern and once all statements have been translated, QUASAR merges all the sub-nets into an unique colored net. We showed in [EKP$^+$05] that colored Petri nets are also able to deal with dynamic aspects of programming languages and especially Ada language and how we have extend QUASAR to allow the verification of Ada program containing dynamic tasks creation or termination.

Quite all patterns we are using for translating Ada programs into colored Petri nets can be reused for translating Promela specifications with the same efficiency. The only specific structure that we have to deal with is communication channels.

Colored Petri nets A colored Petri net may be viewed as a Petri nets with colored (typed) token. Each token is a tuple of values and each place is thus typed by a color domain. Each arc is labeled by variables (or expressions on variables) taking value in the type of the place. When firing a transition, we have first to instantiate variables around this transition (and defined by attaining arcs). Then one checks the presence of these values in places that are pre-conditions of this transition and if the transition is fired, values defined by post-conditions are added to related places.

For instance, in the net presented in left of figure 1, the color domain of place VAR_C is $int \times int$ and the color domain of places $P1$ and $P2$ is int. Each token of the place VAR_C is thus a couple of integer values and each token of places $P1$ and $P2$ is a simple integer value. Transition $T1$ has two entering arcs: one from VAR_C labeled by a tuple of two variables id and val, and one from $P1$ labeled by the variable id. Transition $T1$ can thus be fired when a token of $P1$ has its id value matching any id value of a token in VAR_C. When $T1$ is fired, a new token is instantiated in VAR_C and an other one in $P2$.

From Promela to colored Petri nets We consider now how to map any Promela specification into a colored Petri net. First, to each process of the specification is assigned a unique process identifier id. This identifier is calculated dynamically without introducing any combinatory [EKP+05]. Then, to each variable is associated a place. If the variable is local this place will contain token$<id, val>$ where id is the identifier of the process and val is the value of that variable for this process (if the variable is global then there is only the value of the variable in the place). Then, each statement of the promela specification is translated into a sub-net. When all statements have been translated, all the sub-nets are merged into a single colored Petri net.

Figure 1 presents some patterns illustrating the translation of Promela specification into colored Petri net. The first sub-net is a translation of a simple assignment in which the variable c (modeled by place VAR_C) is incremented by 1. The process identified by id catches its own variable c in the place VAR_C (the token $<id, val>$) and replaces it by a new token $<id, val+1>$. The second sub-net is a translation of a common 'if-then-else' block. The condition is checked by the guards $[val == 0]$ and $[val! = 0]$ on the transitions. If the value of the variable c is equal to 0, the sub-net SUB_THEN is executed. If the value of the variable c is not equal to 0, the sub-net SUB_ELSE is executed.

Modeling the channels In this paper we restrict us to patterns corresponding to channels that are statically created. Thus, the number of channels and their

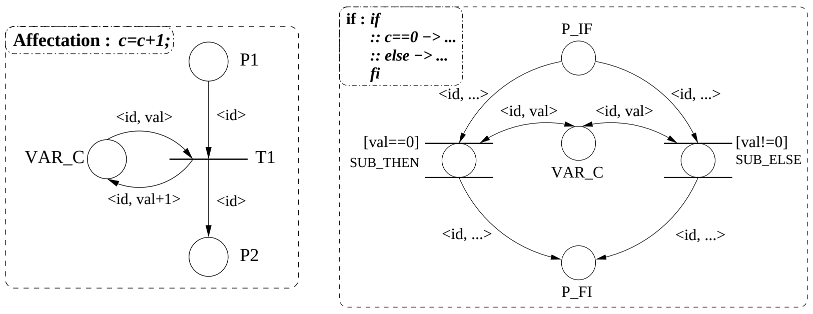

Fig. 1. Some patterns

buffer size are known at the compile time. It is possible to deal with dynamic channel creation but the patterns for channels are quite more complicated.

As a channel with a buffer size equal to 0 is just simple synchronization between the reader and the writer, using a simple transition is thus an instinctive and easy way of modeling. Such a pattern is presented at figure 2. The transition cannot be fired until the writer and the reader have not reached their corresponding `CALL` places. The message exchanged between the two processes is represented by the field `message` in the marks.

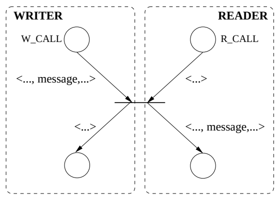

Fig. 2. Pattern for a synchronization channel.

For modeling asynchronous channels we use for each channel (or each array of channels), a unique corresponding place in the net which will contains tokens $<idc$, n, $m>$where idc is the unique identifier instance of the channel, m is the message stored in the entry n of the queue. To the channel place is associated a place which counts the free size of the channel buffer. Furthermore, we use two places (one for the reader, the other for the writer) for catching messages in the FIFO order. The pattern for the asynchronous channel is presented figure 3, where id denotes identifier of processes, idc is the unique identifier of the channel, n is the entry number in the channel queue, m is the message and $SIZE$ is the buffer size of the channel. When a writer wants to write in the channel, it checks if the guarding place (place $FREE$) is not empty. When it is empty, it means that the queue is full and the writer is blocked. When it is not empty, the writer catches the entry number of the queue in the $WRITE$ place and fire the writing transition.

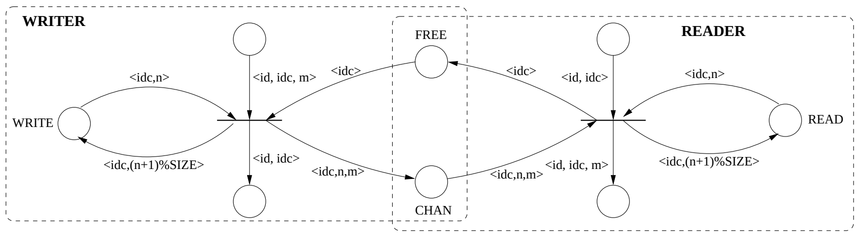

Fig. 3. Pattern for a channel with a buffer size greater than 0.

The firing rule of this transition updates the token of the $WRITE$ place and write the message in the channel by adding a token in the $CHAN$ place. Similarly, when a reader wants to write, it catches the entry number in the $READ$ place, and match the message with the entry number read. When the reader has read the message, it updates the entry number and adds a token in the $FREE$ place for an other writer to write in the channel. In the case we decide to not block a writer when the queue is full we simply suppress place $FREE$. If we want to do not respect the FIFO order, we suppress places $WRITE$ and $READ$.

3.2 A first illustration

Let us consider now the following simple produced / consumer model (presented figure 4). The colored Petri nets (original and reduced) corresponding to this Promela model and build using previous defined patterns are presented in Appendix.

```
int MAX = 10;
chan root = [SIZE] of {int};

proctype reader()
{
    int i;
    int j=1;
    do
       :: (j<=MAX) -> root?i ; j++
       :: (j>MAX) -> break
    od
}
proctype writer()
{
    int j = 1;
    do
       :: (j<=MAX) -> root!j; j++
       :: (j>MAX) -> break
    od
}
init
{
    atomic{
        run writer();
        run reader()
    }
}
```

Fig. 4. A simple producer/consumer written in Promela

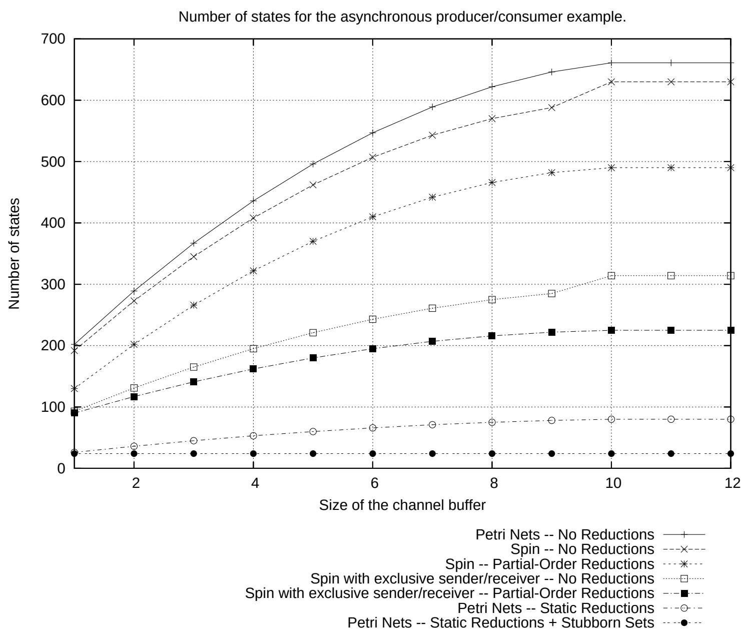

Fig. 5. Number of generated states for the program presented figure 4. `MAX` is set to 10 and the buffer size varies from 1 to 12.

We calculate the number of states generated when the buffer size (modeled by the variable `SIZE`) varies from 1 to 12 and the results are presented in figure 5. We can remark that the number of states generated by Spin when none partial order reductions is used and when no exclusive sender or receiver are specified is quite similar to the number of states generated by the (non reduced) corresponding colored Petri net model. Note that when $SIZE$ is greater or equal to 10 every writing statement can be executed since the writer did not perform more than 10 writing statements. Thus, the number of generated states do not change when the buffer size is greater or equal to the number of loop executed.

We can also remark that the best efficient reduction ratio is obtained when we apply both agglomeration transitions and stubborn sets technique and that the number of states with these techniques is almost from 10 to 20 smaller that the number of states computed with Spin using partial order reduction and exclusive reader/writer channel.

Applying this technique to other Promela specifications leads to comparable results with, some time, a lower ratio between the “best” Spin strategy and the “best” Petri nets strategy. For instance, using the sort algorithm [1] we obtain results showing a reduction factor around 25% (see figure 6).

[1] provided with the Spin distribution and for which Spin partial order method gives remarkable results

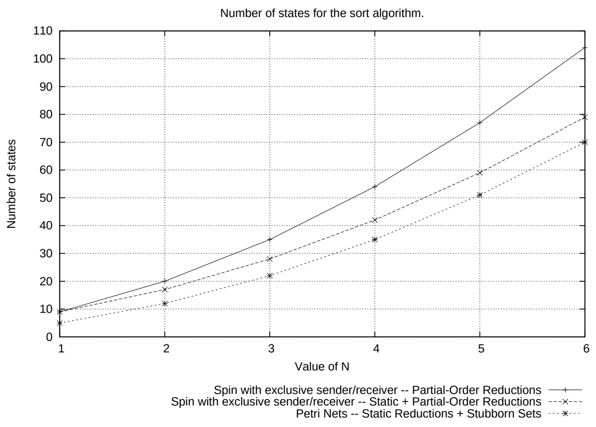

Fig. 6. Evaluations of the sort algorithm.

3.3 Syntactical promela conditions

The main drawback of the strategy used in previous subsection is the need to translate a specification from a formalism to another. We will define now some conditions under which it is possible to automatically infer agglomerations directly in the Promela specification. These agglomerations allow us to group some statements into an atomic block in order to reduce combinatory. In other term we will fix simple conditions that allows us to transform a sequence [2]

$$i_0;\ \texttt{atomic\{}\ i_1;\ i_2;\ \ldots;\ i_k\texttt{\}}$$

into the atomic sequence

$$\texttt{atomic\{}\ i_0;\ i_1;\ i_2;\ \ldots;\ i_k\texttt{\}}$$

The first case we consider is the one where the sequence $s_f = \texttt{atomic\{}\ i_1;\ldots;\ i_k\texttt{\}}$ is a non blocking sequence and where $i_1;\ \ldots;\ i_k$ refer only local variables or constants (i.e. variables that are declared within the corresponding process or that are never assigned except at their declaration). In this case, a post-agglomeration of i_0 with the rest of the sequence can be performed. Indeed, as s_f is non blocking, it can be executed as soon as i_0 has been executed and then the F-continuation hypothesis is fulfilled. Now, as $i_1\ \ldots;\ i_k$ refer only local variables or constants the way the sequences s_f is executable cannot change after the execution of i_0. Indeed, suppose that i_2 refers a variable x and that s_f is executable after i_0 for a value x_0 for x. As x is local, the value of x cannot change before s_f is executed, and then the way of s_f is executed does not change when i_0 has been executed and s_f not. So the F-Independence hypothesis is fulfilled. As statements are executed by a process i_0 cannot be re-executed before s_f has been. So the strongly F-Independence hypothesis is also fulfilled.

[2] the case $k = 0$ is obvious and not studied

Now we distinguish two different cases: i_0 is not the first statement of a sequence of a selection structure (i_0 is not a guard) and i_0 is a guard. Then we examine 3 kinds of statements for i_0 : the blocking conditional statement (`x == y`) the assignment (`x = y`) and the receive operations on a channel (`q?x`) For each of them we determine if its execution can be "delayed" or "advanced" w.r.t. the pre- and the post-agglomeration condition.

The statement i_0 is not a guard Suppose that i_0 is an assignment that do not refer global variable (except constants). As i_0 accesses only variables that cannot change when the process is not active, the statement i_0 can be delayed and then the H-independent and the strongly quasi-persistence hypotheseses are fulfilled. As the statement i_0 is not a loop the divergence freeness is ensured and as we agglomerate a single statement (i_0) with a sequence (s_f) the H-similarity hypothesis is fulfilled ($|I| = 1$). Now, if i_1 is a blocking statement and if i_1 does not use variable modified by i_0 and does not modify variables accessed by i_0 then we can safely replace the statement `atomic{` i_0; i_1; ...; i_k`}` by the statement `atomic{` i_1; i_0; ...; i_k`}`. By this way we put the "blocking" statement at the beginning of the sequence which disable a possible interruption in the atomic statement execution.

Now, suppose that i_0 is a blocking boolean expression and suppose that this expression does not refer to global variables (except constant). Then using same reasoning, a pre-agglomeration can be performed between i_0 and s_f.

When i_0 is a blocking reception on a channel we have to take more precautions. First we have to suppose that the channel is marked as "exclusive reader". This disables the possibility that a process takes a message that another process was waiting for (which will contradict the quasi-persistence hypothesis). Then the H-independence hypothesis implies that the reception of a message on the channel does not enable an action of an other process. In the general case this is not possible (a "reader" can unblock a "writer"). However, suppose that the user can mark a channel as "sufficient capacity" meaning that a writing on this channel will never block. Then, reading a message on such a channel cannot unblock a process waiting for writing. In such a case, a pre-agglomeration can be safely performed.

The statement i_0 is a guard Now suppose that i_0 is the first instruction of a case selection (this applies also to a repetition statement)

```
if
    :: $i_0$; atomic{ $i_1$; ...; $i_k$}
    :: $s_1$
    :: ...
    :: $s_n$
    :: else $s_e$
fi
```

where s_1, ..., s_n and s_e are sequences (atomic or not).

First, suppose that i_0 is an assignment or a boolean expression that uses only local variables or constants. Suppose also that `atomic{` i_1; ...; i_k`}` is a non blocking sequence and that each statement s_j can be written $i_0^j.s'_j$ with s'_j a non blocking sequence and i_0^j an assignment or a boolean expression that uses only local variables or constants. In this case we van perform a pre-agglomeration of i_0 with the sequence `atomic{` $i_1; \ldots; i_k$`}` simultaneously of a pre-agglomeration of each i_0^j with the first statement of s'_j. Indeed, the H-independence and the quasi-persistence are ensured due to the locality of variables used in statements. The H-similarity is obtained by the non blocking character of each sequence s'_j which ensure that if a given sequence s'_j is executable then all other sequences $s'_{j'}$ are also executable.

Second suppose that i_0, is a boolean expression using only local variables and constants. If each statement s_i begins with also a boolean expression using only local variables and constants and if at most one of this boolean expression is true at a time then i_0 can be pre-agglomerated with `atomic{` $i_1; \ldots; i_k$`}`. This is so because there is no really choice on the selection structure : at most one sequence is executable and the one which is executable does not change until it's executed.

A same reasoning can be applied when i_0 is a statement $q?v_0(x_0)$ such that, q is a channel that is marked exclusive reader that does not block writers, and when each s_i is also a statement $q?v_i(x_i)$, where $v_0 \ldots v_n$ design different constant values and where $x_0, \ldots, x_i$ design local variable and when there is no else part in the selection structure. Indeed, in this case, there is no real choice (due to the different value of message type) and as there is no else part, the message reception can be delayed.

At last, suppose that all the alternative of a case statement are atomic sequences. Then, without modifying its behavior we can rewrite it into an atomic sequence that contains the case statement as the unique statement.

Applying these syntactical agglomerations to the producer/consumer example (figure 4) leads to a slightly modified promela model (figure 7) with a smaller state space as depicted by the third curve in figure 8. Comparable results are obtained on the sort model (see the second curve of figure 6).

```
proctype reader()
{
    int i;
    int j=1;
    xr root;
    do
       :: atomic {(j<=MAX) -> root?i ; j++}
       :: (j>MAX) -> break
    od
}
proctype writer()
{
    int j = 1;
    xs root;
    do
       :: atomic{(j<=MAX) -> root!j; j++}
       :: (j>MAX) -> break
    od
}
```

Fig. 7. The simple producer/consumer with automatically inferred atomic blocks.

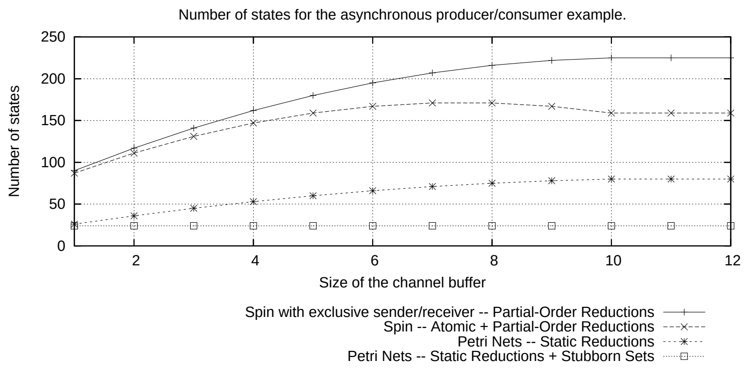

Fig. 8. Number of generated states for the program presented figure 4. `MAX` is set to 10 and the buffer size varies from 1 to 12.

4 Related works on syntactical model reductions

First works concerning reduction of sequences into atomic actions for simplification purpose was performed by Lipton in [Lip75]. Lipton focused only on deadlock property preservation. Using parallel program notations of Dijkstra he defined "left" and "right" movers. Roughly speaking, a "left" (resp. "right") mover is a local process statement that can be moved forward (resp. delayed) w.r.t. statements of others processes without modifying the halting property. Lipton then demonstrated that, in principle, the statement `P(S)`, where `S` is a semaphore, is a "left" mover and `V(s)` is a "right" mover. Then Lipton proved that some parallel program are deadlock free by moving `P(S)` and `V(S)` statements and by suppressing atomic statements that have no effect on variables. However, two difficulties arise: the reduction preserves only deadlocks and the application conditions are difficult to be checked.

Cohen and Lamport propose in [CL98] assumptions on TLA specifications under which they define a reduction theorem preserving liveness and safety properties. This work fixes the reduction theorem in a "high" level formalism which can be a clear advantage for defining specific utilization. However, it's also its main drawback since it is based on the hypothesis that some actions commute, but no effective way is proposed to check whether this assumption holds.

More recently, Cohen, Stoller, Qadeer, and Flanagan [SC03], [FQ03b], [FQ03a] leveraged Lipton's theory of reduction to detect transactions in multi-threaded programs (and consider these transactions as atomic actions in the model checking step). Stoller and Cohen propose in [SC03] a reduction theorem based on omega algebra that can be applied to models of concurrent systems using mutual exclusion for access to selected variables. However, they use a restricted notion of "left" mover and a better reduction ratio can be obtained by applying more accurate reductions (as demonstrated in [EHPP04a]). Moreover, their reductions are justified by the correct use of "exclusive access predicates" and by the respect of a specific synchronization discipline. These predicates may be difficult to compute and no effective algorithm is given to test that the synchronization discipline is respected.

Flanagan and Qadeer noted in [FQ03a] that the previous authors use only the notion of "left" mover and proposed an algorithm that uses both "left" and "right" mover notions to infer transactions. However, this algorithm is based on access predicates that can be automatically inferred only for specific programs using lock-based synchronization. Moreover, as they use both "left" and "right" movers to obtain a better reduction ratio and as they do not fix sufficient restrictive application conditions, their reduction theorem do not preserve deadlock.

In Petri nets formalism, the first works concerning reductions have been performed by Berthelot [Ber85]. The link between transition agglomerations (the most effective structural reductions proposed by Berthelot) and general properties, expressed in LTL formalism, is done in [PPP00].

In [ES01], Esparza and Schröter simplify one point in the original pre-agglomeration conditions. However, they consider only 1-safe Petri nets (each place is bounded by 1), the application conditions remain purely structural, and as the authors focus only on infinite sequences preservation, their reductions do not even preserve the deadlock property! [3]

We proposed in [HPP04] new Petri nets reductions that cover a large range of patterns by introducing algebraic conditions whereas the previously defined ones rely solely on structural conditions. We adapted them in [EHPP04b,EHPP04a] to colored Petri nets which are an abbreviation of Petri nets and define a concise formalism for the modeling of concurrent software. We showed here that these reductions can also be adapted to Promela specifications leading to simple syntactical rules which permit a significant reduction of the combinatory while preserving properties of the model.

5 Conclusion

We showed in this paper that efficient Petri nets reductions can be used to significantly reduce the state space size of a Promela specification. We proposed two approaches among which one is based on simple syntactical rules allowing the automatic building of atomic sequences. We showed on classical examples the efficiency of these approaches. We are currently working on the implementation of this rules in order to confirm the benefit of the approach on large Promela models.

References

[AHI98] K. Ajami, S. Haddad, and J-M. Ilié. Exploiting symmetry in linear time temporal logic model checking: One step beyond. *Lecture Notes in Computer Science*, 1384, 1998.

[Ber83] G. Berthelot. *Transformation et analyse de réseaux de Petri, applications aux protocoles.* Thèse d'état, Université Pierre et Marie Curie, Paris, 1983.

[3] Note moreover that being 1-safe is not a stable characteristic w.r.t. reductions.

[Ber85] G. Berthelot. Checking properties of nets using transformations. In G. Rozenberg, editor, *Advances in Petri nets*, volume No. 222 of *LNCS*. Springer-Verlag, 1985.

[CL98] Ernie Cohen and Leslie Lamport. Reduction in TLA. In *International Conference on Concurrency Theory*, pages 317–331, 1998.

[EHPP04a] S. Evangelista, S. Haddad, and J.F. Pradat-Peyre. Coloured Petri nets reductions for concurrent software validation. Technical report, CEDRIC, CNAM, Paris, 2004.

[EHPP04b] S. Evangelista, S. Haddad, and J.F. Pradat-Peyre. New coloured reductions for software validation. In *Workshop on Discrete Event Systems*, 2004.

[EKP$^+$05] S. Evangelista, C. Kaiser, C. Pajault, J. F. Pradat-Peyre, and P. Rousseau. Dynamic tasks modeling for concurrent programs verification with QUASAR. In *Reliable Software Technologies - Ada-Europe 2005*, LNCS. Springer-Verlag, 2005.

[EKPPR03] S. Evangelista, C. Kaiser, J. F. Pradat-Peyre, and P. Rousseau. Quasar: a new tool for analysing concurrent programs. In *Reliable Software Technologies - Ada-Europe 2003*, volume 2655 of *LNCS*. Springer-Verlag, 2003.

[EPS93] A.E. Emerson and A. Prasad Sistl. Symmetry and model checking. In *proc. of the 5th conference on Computer Aided Verification*, June 1993.

[ES01] J. Esparza and C. Schröter. Net Reductions for LTL Model-Checking. In T. Margaria and T. Melham, editors, *Correct Hardware Design and Verification Methods (CHARME'01)*, volume 2144 of *Lecture Notes in Computer Science*, pages 310–324. Springer-Verlag, 2001.

[FQ03a] Cormac Flanagan and Shaz Qadeer. Transactions for software model checking. In Byron Cook, Scott Stoller, and Willem Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[FQ03b] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349. ACM Press, 2003.

[GW93] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Form. Methods Syst. Des.*, 2(2):149–164, 1993.

[HPP04] S. Haddad and J.F. Pradat-Peyre. Efficient reductions for LTL formulae verification. Technical report, CEDRIC, CNAM, Paris, 2004.

[Lip75] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[PPP00] D. Poitrenaud and J.F. Pradat-Peyre. Pre and post-agglomerations for *LTL* model checking. In M. Nielsen and D Simpson, editors, *High-level Petri Nets, Theory and Application*, number 1825 in LNCS. Springer-Verlag, 2000.

[SC03] Scott D. Stoller and Ernie Cohen. Optimistic synchronization-based state-space reduction. In H. Garavel and J. Hatcliff, editors, *TACAS'03*, volume 2619 of *Lecture Notes in Computer Science*. Springer-Verlag, April 2003.

[Sis03] A. Prasad Sistla. Symmetry reductions in model-checking. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, London, UK, 2003. Springer-Verlag.

[Val93] Antti Valmari. On-the-fly verification with stubborn sets. In *Proceedings of the 5th International Conference on Computer Aided Verification*, pages 397–408. Springer-Verlag, 1993.

[WG93] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In E. Best, editor, *CONCUR'93: Proc. of the 4th International Conference on Concurrency Theory*, pages 233–246. Springer, Berlin, Heidelberg, 1993.

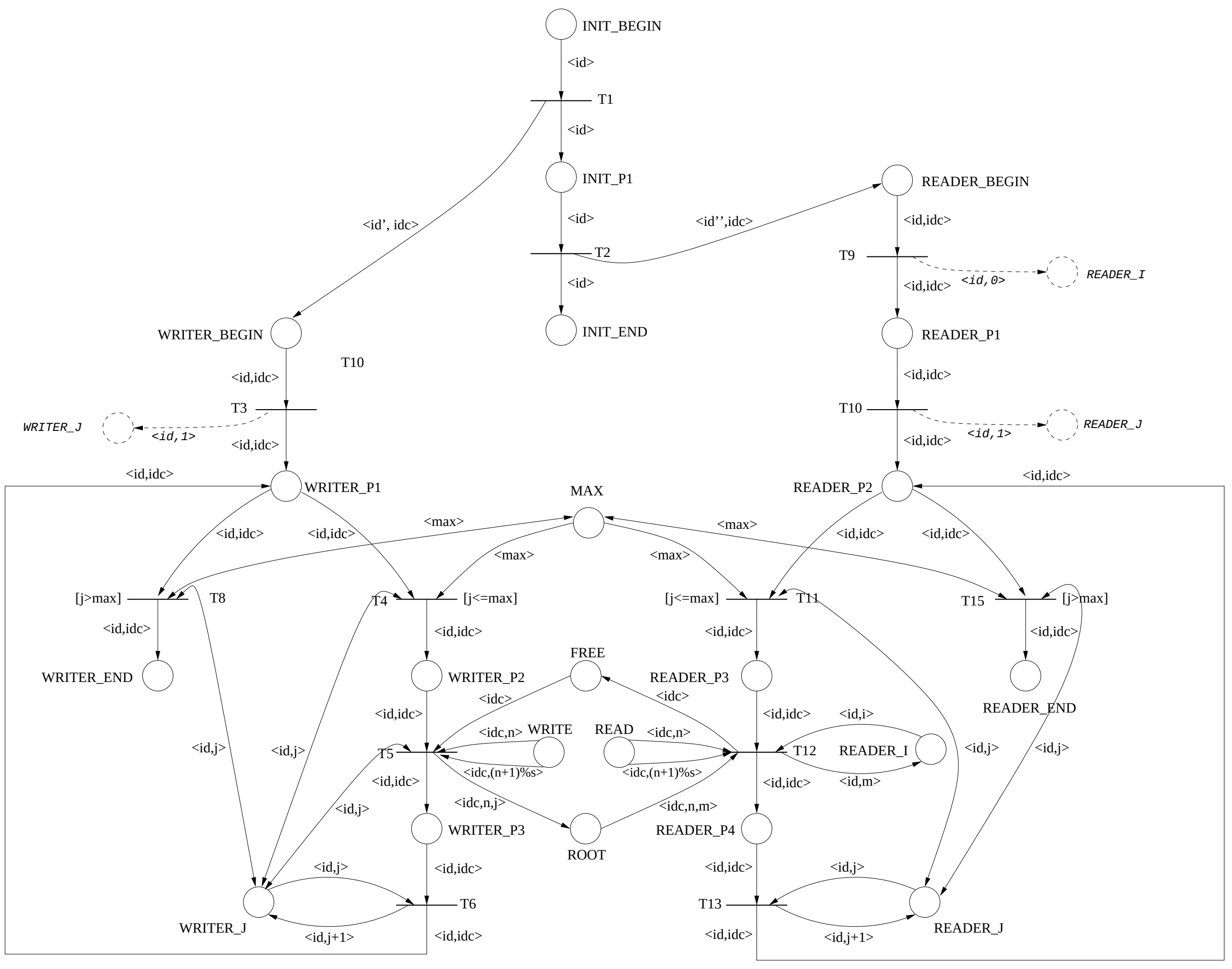

Fig. 9. The colored Petri net of the Promela specification depicted figure 4

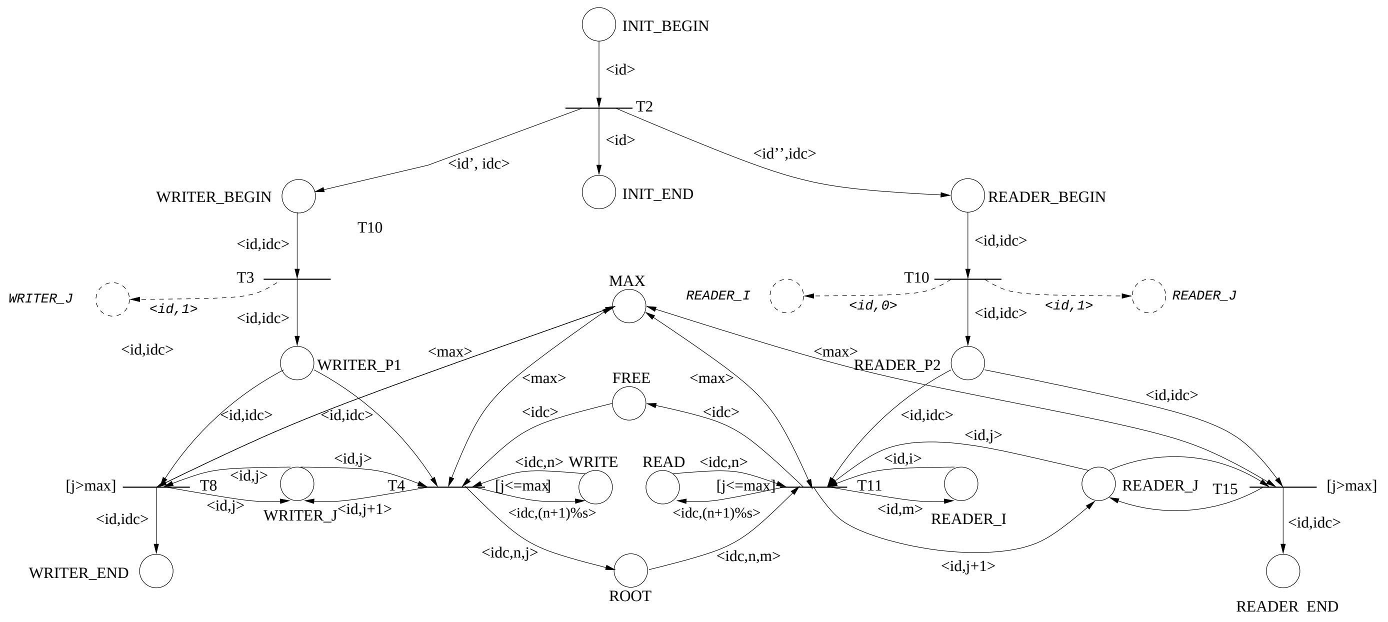

Fig. 10. The reduced colored Petri net of the Promela specification depicted figure 4

```
/*
 * A program to sort concurrently N "random" numbers
 * The reduced space and time should be linear in the
 * number of processes, and can be reduced when the length of
 * the buffer queues is increased.
 * In full search it should be exponential.
 */

#define N       7                       /* Number of Proc */
#define L       10                      /* Size of buffer queues */
#define RANDOM  (seed * 3 + 14) % 100   /* Calculate "random" number */

chan q[N] = [L] of {byte};

proctype left(chan out)         /* leftmost process, generates random numbers */
{       byte counter, seed;

        xs out;

        counter = 0; seed = 15;
        do
        :: out!seed ->                  /* output value to the right */
                atomic{
                counter = counter + 1;
                atomic{
                if
                :: atomic{counter == N -> break}
                :: atomic{counter != N -> skip}
                fi;
                }
                seed = RANDOM           /* next "random" number */
                }
        od
}

proctype middle(chan in, out; byte procnum)
{       byte counter, myval, nextval;

        xs out;
        xr in;

        counter = N - procnum;
        in?myval;                       /* get first value from the left */
        do
        :: counter > 0 ->
                atomic{
                in?nextval;             /* upon receipt of a new value */
                atomic{
                if
                :: atomic{nextval >= myval -> out!nextval}
                :: atomic{nextval <  myval ->
                        out!myval;
                        myval=nextval}  /* send bigger, hold smaller */
                fi;
                }
                counter = counter - 1
                }
        :: counter == 0 -> break
        od
}

proctype right(chan in) /* rightmost channel */
{       byte biggest;

        xr in;

        in?biggest      /* accepts only one value, which is the biggest */
}

init {
        byte proc=1;

        atomic {
                run left ( q[0] );
                do
                :: proc < N ->
                        run middle ( q[proc-1] , q[proc], proc );
                        proc = proc+1
                :: proc == N -> break
                od;
                run right ( q[N-1] )
        }
}
```

Fig. 11. The sort algorithm modified