Adapting Petri Nets Reductions to Promela Specifications

C. Pajault[1], J.-F. Pradat-Peyre[1], and P. Rousseau[2]

[1] LIP6, Université Pierre et Marie Curie, Paris,
{Christophe.Pajault,Jean-Francois.Pradat-Peyre}@lip6.fr
[2] Cedric-CNAM, rousseau@cnam.fr

Abstract. The interleaving of concurrent processes actions leads to the well-known combinatorial explosion problem. Petri nets theory provides some structural reductions to tackle this phenomenon 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 does not observe the modified transitions. Furthermore, they can be combined with other kinds of reductions such as partial-order techniques to improve the efficiency of state space reduction. We present in this paper an adaptation of these reductions for Promela specifications and propose simple rules to automatically infer atomic steps in the Promela model while preserving the checked property. We demonstrate on typical examples the efficiency of this approach and propose some perspectives of this work in the scope of software model checking.

1 Introduction

The interleaving of concurrent processes actions leads to a combinatory explosion. In order to give a simple insight of this problem, let us consider a simple 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 [1], sleep sets [2], stubborn sets [3], ...), or symmetry based reductions [4, 5] 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. The goal of a reduction theory is to (syntactically) characterize situations where a reduction is sound and how to perform it.

Based on this principle, we proposed in [6] some new Petri nets reductions that cover a large range of synchronization patterns. We extended these reductions to colored Petri nets (which are an abbreviation of Petri nets) in [7, 8] and use them in the Quasar [9] platform that performs verification of concurrent Ada programs by analyzing an intermediate colored Petri net generated from a given program.

These reductions yield very interesting results and we present in this paper how these reductions can be adapted to simplify program analysis without needing a translation step into a more formal model (such as Petri nets). We illustrate our approach with the Promela language since it's a simple and clear language associated to the very efficient model checker Spin [10].

More precisely, we define some syntactical rules based on Petri nets agglomerations which allow the automatic detection of sequences of statements that can be marked as "atomic" (using the `atomic` construction of Promela) while preserving analyzed properties. The interest of this transformation is to significantly reduce interleaving and thus the size of the state space.

2 Petri nets transitions agglomerations

A Petri net reduction is characterized by some application conditions, a net transformation and 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 pre- and the post-agglomerations, we briefly recall some Petri nets definitions.

2.1 Brief Petri nets definitions and notations

Definition 1 (Petri net model). *A marked net (N, m_0) is defined by 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. The transitions linked to a place p are defined by $^\bullet p = \{t | W^+(p,t) > 0\}$ and $p^\bullet = \{t | W^-(p,t) > 0\}$.*

Definition 2 (Firing rule). *Let (N, m_0) be a marked net then a transition $t \in T$ is firable from a marking m (denoted by $m[t\rangle$) 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 by $m[t\rangle m'$) defined by $\forall p \in P$ $m'(p) = m(p) + W(p,t)$ where W the incidence matrix is defined by $W = W^+ - W^-$. A marking m such that $\forall t \in T$, $NOT(m[t\rangle)$ is called a* **dead** *marking.*

We use the following notations.

- T^* is the set of finite sequences of transitions and T^ω is the set of infinite sequences of transitions; λ defines the empty sequence of transitions;
- $\Pi_{T'}(s)$ 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;
- $Pref(s) = \{s' \,|\, \exists s''$ s.t. $s = s'.s''\}$ denotes the set of prefixes of s.

Definition 3 (Firing rule extension). *Let (N, m_0) be a marked net. A finite sequence $s \in T^*$ is firable from m, a marking and leads to m' (also denoted by $m[s\rangle$ and $m[s\rangle m'$) iff either $s = \lambda$ and $m' = m$ or $s = s_1.t$ with $t \in T$ and $\exists m_1\ m[s_1\rangle m_1$ and $m_1[t\rangle m'$ We note $Reach(N, m_0) = \{m | \exists s \in T^*\ m_0[s\rangle m\}$ the set of reachable markings. An infinite sequence $s \in T^\omega$ 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 (Generated language). *Let (N, m_0) be a marked net then*

- $L(N, m_0) = \{s \in T^* | m_0[s\rangle\}$ *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,*
- $L^\omega(N, m_0) = \{s \in T^\omega | m_0[s\rangle\}$ *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 behavior 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 behavior w.r.t. to a set of properties. This restricted behavior is the behavior of a reduced net, denoted (N_r, m_0), defined in Appendix.

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: $\forall t \in T_0, \phi(t) = t$ 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 $\mathcal{P}(T^\infty)$.

In order to obtain the preservation properties (such like deadlock occurrences) we have to introduce behavioral hypotheseses. The basic one, named *Potential agglomerability* ensures that an occurrence of a transition of F is always preceeded by an occurrence 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 5 (P-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) directly on the Petri net.

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

Definition 6. *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 firable 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 [6].

Theorem 1. *If a p-agglomerable Petri net (N, m_0) is also*

1. *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. *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. *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 behavioral 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 modeling point of view, the set F represents local actions while the set H corresponds to global actions possibly involving synchronization.

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

1. *F**-independent** iff $\forall i \in I$, $\forall h \in H_i$, $\forall f \in F_i$, $\forall s \in (T_0 \cup H)^*$, $\forall m \in Reach(N, m_0)$, $m[h.s.f\rangle \Longrightarrow m[h.f.s\rangle$*
   *(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. $\forall s' \in Pref(s), \Gamma(s') \geq 0$ $\forall m \in Reach(N, m_0)$, $m[h.s.f\rangle \Longrightarrow m[h.f.s\rangle$*
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 [6]).

Theorem 2. *If a p-agglomerable Petri net (N, m_0) is also*

1. *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. *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 Simplifying Promela model analysis

3.1 The Promela language

Promela is a verification modeling language associated with the Spin tool. Promela specifications consist of processes, message channels and variables. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior while channels and global variables define the processes environment.

The execution of every statement is conditional on its executability. Statements are either executable or blocked. For instance, an assignment `j=1` is always executable while a boolean condition `j==1` is *executable* only when `j` is equal to 1.

An important feature of Promela is the ability given to processes to synchronize themselves through message channels which are used to model the transfer of data from one process to another. They are declared either locally or globally.

```
 1 #define SIZE 255
 2
 3 int N = 100;
 4 chan root = [SIZE] of {int};
 5
 6 proctype reader()
 7 {
 8     int i;
 9     int j = 1;
10     do
11        :: ( j<= N ) -> root?i; j++
12        :: ( j > N ) -> break
13     od
14 }
15 proctype writer()
16 {
17     int j = 1;
18     do
19        :: ( j<= N ) -> root!j; j++
20        :: ( j > N ) -> break
21     od
22 }
23 init
24 {
25     atomic{
26         run writer();
27         run reader()
28     }
29 }
```

Fig. 1. A simple producer/consumer Promela model

Figure 1 depicts an example of Promela specification. This simple producer/consumer example spawns two processes: one of type `writer` and one of type `reader`. The writer sends `N` messages to the reader via a channel.

On line 4, `chan root = [ SIZE ] of { int };` declares a channel that can store up to `SIZE` messages of type `int`. The statement `root!j` (line 19) is a transmission of the value of `j` on the channel `root` (i.e. it appends the value to the tail of the channel). And `root?i` (line 11) models the reception on that channel (i.e. it retrieves it from the head of the channel, and stores it in the variable `i`). The send operation is executable only when the channel addressed is not full. The receive operation, similarly, is only executable when the channel is non empty. The channels pass messages in a first-in-first-out order.

The control flow in Promela can be defined with the selection, the repetition, and the unconditional jumps. For instance, The lines 18-21 in Figure 1 contains a repetition statement (`do ... od`). This repetition statement contains two sequences of statements, each preceded by a double colon. The first statements of these execution sequences are called *guards*. This repetition sequence will either execute the sequence starting with `:: ( j <= N )` or the sequence starting with `:: ( j > N )` regarding which guard is executable. If several guards are executables, one is randomly chosen. Once the sequence is executed, the repetition statement will be repeated. The normal way to terminate the repetition structure is the use of the `break` statement. The selection statement is similar to the repetition statement, but occurs only once.

3.2 Syntactical Promela agglomerations

We define now some conditions under which it is possible to automatically infer agglomerations in a Promela specification. These agglomerations group sequential statements into an atomic block in order to reduce the combinatory. In other term we will fix simple conditions that allow us to transform a sequence [3]

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

into the atomic sequence

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

We study different cases for which this transformation preserves deadlock, non progress cycles or any LTL formula that does not observe the action i_0 (when we perform a pre-agglomeration) or any actions of i_j (when performing a post-agglomeration).

Methodology used The main principle is to explicitly use the behavioral conditions of Petri nets agglomerations for characterizing different cases in which the transformation is correct w.r.t. the analyzed property. Indeed, for any Promela specification, a corresponding Petri net with the same behavior can be generated (as in the QUASAR project for analyzing concurrent dynamic Ada programs) and adapted to Promela specification in [11]. Then, agglomeration conditions on Petri nets can be translated into syntactical and semantical conditions in the Promela program.

In order to interpret Petri net behavior into Promela behavior we classify Promela statements following three criteria (in the following examples we suppose that a statement i_0 is followed by a sequence of statement $s_f = \{i_1 \ldots;\ i_k\})$:

1. *is the statement blocking or not?* For instance, an assignment is a non blocking statement, because it is always executable while a boolean expression or a receive operation on a channel are blocking statements; this characteristic is related to the F-continuation hypothesis ;
   Indeed, if the statement i_0 is followed by a sequence of non blocking statements s_f, we know that the sequence will always be executed if i_0 is executed and so the F-continuation hypothesis is fulfilled (in the Petri net model, all the transitions modeling the sequence execution will be executable after firing i_0).

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

2. *does the statement refers local or global variables?* When the statement refers only local variables, the value for which the statement is executed by a process cannot change by the execution of other processes ; this characteristic is related to the H- and the F-independence ;
   For s_f referring only local variables or constants, the way the sequence s_f is executable cannot change after the execution of i_0. More precisely, suppose that i_2 refers a variable x and that s_f is executable after i_0 for a value x_0 of x ; as x is local, the value of x cannot change after i_0 is executed and before s_f execution, and then the way that s_f is executed does not change. So the F-Independence condition is fulfilled. As statements are executed by a process, i_0 cannot be re-executed before s_f has been executed. The strongly F-Independence condition is then also fulfilled (in the Petri net model, it would mean that the transitions of the sequence do not access places also accessed by a transition modeling another process execution).
   The same reasoning can be performed for i_0, but in this case would fulfill the H-independent and the strongly quasi-persistence conditions. 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 conditions are fulfilled.

3. *is the statement a guard (a first statement of a sequence in a branch of a selection)?* in that case, the statement is potentially in competition with other statements in other branches of the selection structure (the process may have a choice); this characteristic is related to the quasi-persistent and the H-similar hypothesizes.
   If there is no competition (the statement is not a guard) the H-similarity condition is fulfilled ($|I| = 1$). Others cases have to be discussed for each kind of statement.

The statement i_0 is followed by non blocking statements The first case is when the sequence $s_f =$ `atomic{` i_1;. . .; i_k`}` is a non blocking sequence and $i_1; \ldots;$ i_k only refers local variables or constants (i.e. variables that are declared within the corresponding process or that are never assigned except at their declaration). In that 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 condition is fulfilled. Now, as $i_1 \ldots;$ i_k refers only local variables or constants the way the sequences s_f is executable cannot change after the execution of i_0. More precisely, suppose that i_2 refers a variable x and that s_f is executable after i_0 for a value x_0 of x; as x is local, the value of x cannot change before s_f is executed, and then the way that s_f is executed does not change when i_0 has been executed and s_f not. So the F-Independence condition is fulfilled. As statements are executed by a process, i_0 cannot be re-executed before s_f has been executed. The strongly F-Independence condition is then also fulfilled.

The statement i_0 is not a guard We suppose here that i_0 is not a guard and we examine three kinds of statements for i_0 : the blocking conditional statement (`x == y`) the assignment (`x = y`) and the receive operation on a channel (`q?x`)

1. Suppose first that i_0 is an assignment that does not refer to global variables (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 conditions are fulfilled. As the statement i_0 is not a loop, the divergence freeness is ensured. Moreover, as we agglomerate a single statement (i_0) with a sequence (s_f), the H-similarity condition is fulfilled ($|I| = 1$). Now, if i_1 is a blocking statement and if i_1 does not use variables modified by i_0 and does not modify variables accessed by i_0, 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 disables a possible interruption in the atomic statement execution.
2. Now, suppose that i_0 is a boolean expression and suppose that this expression does not refer to global variables (except constants). Then using the same reasoning, a pre-agglomeration can be performed on i_0 and s_f.
3. When i_0 is a blocking reception on a channel we have to be careful. First, 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 condition). Then the H-independence condition implies that the reception of a message on the channel does not enable any 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 statement on this channel will never be blocked; 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 statement of a selection structure (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 of actions (atomic or not).

1. 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, that each statement s_j can be written $i_0^j.s'_j$ with s'_j a non blocking sequence and that i_0^j is an assignment or a boolean

expression that uses only local variables or constants; then we can perform a pre-agglomeration of i_0 with the sequence `atomic{` $i_1; \ldots; i_k$`}` simultaneously with 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 ensures that if a given sequence s'_j is executable, then all other sequences $s'_{j'}$ are also executable.

2. Next, suppose that i_0 is a boolean expression using only local variables and constants. If all statements s_i also begin with 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 is executed.
3. The same reasoning can be applied when i_0 is a statement $q?v_0(x_0)$ such that q is a channel marked as an exclusive reader which does not block writers under the conditions that:
   (a) each s_i is also a statement $q?v_i(x_i)$, where $v_0 \ldots v_n$ denotes different constant values,
   (b) $x_0, \ldots, x_i$ design local variables and
   (c) there is no *else* part in the selection structure.

   Indeed, in that 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.
4. At last, suppose that all alternatives of a selection statement are atomic sequences; then, without modifying its behavior we can rewrite it into an atomic sequence that contains the selection statement as the unique statement.

The following algorithm (Algorithm 1) formalizes these Promela sources transformation rules.

4 Experimentations

We implemented these agglomerations in a tool (atomicSpin [12]). We applied our transformations on different models. Using the Spin tool (v. 4.3.0), we compute the total number of generated states when looking for all invalid end-states in the original and in the reduced model.

Consider the producer/consumer specification depicted on Figure 1. The resulting specification after agglomerations is depicted on Figure 2.

Consider process of type `Reader`: the sequence `root?i; j++` can be transformed into `atomic {root?i; j++}` using the first rule (3.1.2) which corresponds to a post-agglomeration. Indeed, `j++` is a non blocking statement using only local variable. Then, we can operate the transformation of the sequence `(j <= N); atomic {root?i; j++}` into the sequence

Algorithm 1 atomicSpin algorithm

```
Require: $i_0$, $i_1$, $s$
  if atomic sequence $s$ doesn't use global variables then
    if atomic sequence $s$ doesn't contain any blocking statement then
      if $i_0$ is not a guard then
        add $i_0$ to the atomic sequence $s$
      else if $i_0$ is an assignment, printf, or general statement then
        add $i_0$ to the atomic sequence $s$
      else if $i_0$ is a boolean expression and
             all other choices are boolean expression and
             at most one choice is true at a time
       then
        add $i_0$ to the atomic sequence $s$
      else if $i_0$ is a channel reception and
             channel is exclusive reader and
             all choices are receptions on the same channel and
             there is no else part in the selection statement and
             there is only one reception computable at a time
       then
        add $i_0$ to the atomic sequence $s$
      else
        close the atomic sequence $s$
      end if
    else
      if $i_0$ is a guard then
        close the atomic sequence $s$
      else if $i_0$ is an assignment or a boolean expression and
             $i_0$ doesn't modify $i_1$ variables and
             $i_0$ access only local variables
       then
        add $i_0$ to the atomic sequence $s$
        swap $i_0$ and $i_1$
      else if $i_0$ is a reception on a channel and
             the channel is exclusive reader and
             the channel is sufficient capacity
       then
        add $i_0$ to the built atomic sequence
        swap $i_0$ and $i_1$
      else
        close the atomic sequence $s$
      end if
    end if
  else
    close the atomic sequence $s$
  end if
```

```
#define SIZE 255

int N = 100;
chan root = [SIZE] of {int};

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

Fig. 2. The simple producer/consumer with automatically inferred atomic blocks.

`atomic{ (j <= N); root?i; j++}` using the second rules of subsection 3.1.4. Symmetric transformations can be applied in the code of the `Writer` process.

In the Table 1, we trace the number of reachable states for the original and the transformed model using the Spin partial order reduction in both cases (SPO means Spin with partial order reductions, and APO means atomicSpin with the same partial order reductions). Agglomerations leads to a significant reduction of the state space size, with a quasi null cost in time (checking our conditions on Promela programs is a very simple task).

We also carried out experiments on classical examples of Promela models included in the standard distribution of Spin (the leader election, the distributed sort, the two versions of a cell-phone handoff strategy in a mobile network, the Peterson for N, the snooping cache) and some examples of our own (two version of the well known dining philosophers and a task allocator modeling a client-server application).

Agglomerations improve the state space reduction in all cases. Except for the leader election, the reduction factor is at least 2 even when partial-order reductions are enabled and agglomerations performed on the dining philosophers model achieved reductions of the state space by a factor 20.

5 Related works on syntactical model reductions

First works concerning reduction of sequences into atomic actions for simplification purpose was performed by Lipton in [13]. 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

Table 1. Benchmarks

name	process	states		memory (Mo)	
		SPO	APO	SPO	APO
allocator	1	44	23	0	0
	2	1 818	646	0	0
	3	42 419	11 876	6	2
	4	637 398	160 242	107	26
	5	7.77e+06	1.89e+06	1 555	378
leader	20	367	267	0	0
	40	727	527	1	1
	60	1 087	787	2	2
	80	1 447	1 047	4	3
	100	1 807	1 307	7	5
leader2	2	84	60	0	0
	3	356	254	0	0
	4	2 074	1 482	0	0
	5	14 122	10 082	3	2
	6	106 514	75 986	29	21
petersonN	3	2 999	2 374	0	0
	4	533 083	383 478	21	21
	5	-	-	-	-
philo1	2	42	20	0	0
	4	1 525	236	0	0
	6	64 944	3 745	6	0
	8	2.91e+06	62 712	373	8
	10	-	1.02e+06	-	155
philo2	10	189 445	86 407	20	9
	11	706 565	292 125	84	35
	12	2.61e+06	955 822	334	122
	13	9.50e+06	3.14e+06	1 293	427
	14	-	1.02e+07	-	1 478
prod_cons	200	161 609	41 009	172	44
	400	558 529	214 789	596	229
	600	967 329	419 589	1 033	448
	800	1.37e+06	624 389	1 469	666
	1000	1.78e+06	829 189	1 906	885
sort	50	5 252	2 705	7	3
	100	20 502	10 404	51	26
	150	45 752	23 104	172	87
	200	81 002	40 804	406	204
	250	126 252	63 304	791	398

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 [14] 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 [15], [16], [17] 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 [15] 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 [18]). 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 [17] 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 [19]. The link between transition agglomerations (the most effective structural reductions proposed by Berthelot) and general properties, expressed in LTL formalism, is done in [20].

We proposed in [6] 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 [7] 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.

6 Conclusion

We demonstrate in this paper that efficient Petri nets reductions can be used to significantly reduce the state space size of a Promela specification. We propose simple syntactical rules allowing the automatic building of atomic sequences. Our experiments highlight the efficiency of these approaches. A first implementation of these rules has already been developed [12]. Our experience with Petri nets allows us to expect even better reductions for more complex models. Based on these experimentations we plan to adapt these transformation rules to automatically infer transactions in concurrent software written in Ada or Java in the near future.

References

1. Wolper, P., Godefroid, P.: Partial-order methods for temporal verification. In Best, E., ed.: CONCUR'93: Proc. of the 4th International Conference on Concurrency Theory. Springer, Berlin, Heidelberg (1993) 233–246
2. Godefroid, P., Wolper, P.: Using partial orders for the efficient verification of deadlock freedom and safety properties. Form. Methods Syst. Des. **2**(2) (1993) 149–164
3. Valmari, A.: On-the-fly verification with stubborn sets. In: Proceedings of the 5th International Conference on Computer Aided Verification, Springer-Verlag (1993) 397–408
4. Emerson, A., Prasad Sistl, A.: Symmetry and model checking. In: proc. of the 5th conference on Computer Aided Verification. (June 1993)
5. Sistla, A.P.: Symmetry reductions in model-checking. In: VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation, London, UK, Springer-Verlag (2003)
6. Haddad, S., Pradat-Peyre, J.: Efficient reductions for LTL formulae verification. Technical report, CEDRIC, CNAM, Paris (2004)
7. Evangelista, S., Haddad, S., Pradat-Peyre, J.: New coloured reductions for software validation. In: Workshop on Discrete Event Systems. (2004)
8. Haddad, S., Pradat-Peyre, J.F.: New efficient petri nets reductions for parallel programs verification. Parallel Processing Letters **16**(1) (2006) 101–116
9. Evangelista, S., Kaiser, C., Pradat-Peyre, J.F., Rousseau, P.: Quasar: a new tool for analysing concurrent programs. In: Reliable Software Technologies - Ada-Europe 2003. Volume 2655 of LNCS., Springer-Verlag (2003)
10. Holzmann, G.J.: The model checker SPIN. Software Engineering **23**(5) (1997) 279–295
11. Pajault, C., Pradat-Peyre, J.: Static reductions for promela specifications. Technical Report 1005, Conservatoire National des Arts et Métiers, laboratoire Cedric, Paris, France (2006)
12. : http://quasar.cnam.fr/atomicSpin/
13. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM **18**(12) (1975) 717–721
14. Cohen, E., Lamport, L.: Reduction in TLA. In: International Conference on Concurrency Theory. (1998) 317–331

15. Stoller, S.D., Cohen, E.: Optimistic synchronization-based state-space reduction. In Garavel, H., Hatcliff, J., eds.: TACAS'03. Volume 2619 of Lecture Notes in Computer Science., Springer-Verlag (April 2003)
16. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, ACM Press (2003) 338–349
17. Flanagan, C., Qadeer, S.: Transactions for software model checking. In Cook, B., Stoller, S., Visser, W., eds.: Electronic Notes in Theoretical Computer Science. Volume 89., Elsevier (2003)
18. Evangelista, S., Haddad, S., Pradat-Peyre, J.: Coloured Petri nets reductions for concurrent software validation. Technical report, CEDRIC, CNAM, Paris (2004)
19. Berthelot, G.: Checking properties of nets using transformations. In Rozenberg, G., ed.: Advances in Petri nets. Volume No. 222 of LNCS., Springer-Verlag (1985)
20. Poitrenaud, D., Pradat-Peyre, J.: Pre and post-agglomerations for *LTL* model checking. In Nielsen, M., Simpson, D., eds.: High-level Petri Nets, Theory and Application. Number 1825 in LNCS, Springer-Verlag (2000)

A Agglomerated Petri net definition

Proposition 1. *The incidence matrices W, W^- and W^+ can be extended to matrices indexed by $P \times T^*$ by the following recursive definition:*

- $W(p,\lambda) = W^-(p,\lambda) = W^+(p,\lambda) = 0$ *and* $W(p, s_1.t) = W(p, s_1) + W(p, t)$
- $W^-(p, s_1.t) = Max(W^-(p, s_1), W^-(p, t) - W(p, s_1))$
- $W^+(p, s_1.t) = W(p, s) + W^-(p, s)$

such that this extension is compatible with the firing rule, i.e.
$\forall s \in T^*$, $m[s\rangle m' \iff \forall p \in P,\ m(p) \geq W^-(p, s)$ *and* $m'(p) = m(p) + W(p, s)$

Definition 8 (Reduced net). *The reduced Petri net (N_r, m_0) 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* $W_r^+(p, hf) = W^+(p, h.f)$