Verification of a hierarchical generic mutual exclusion algorithm

Souheib Baarir[1] and Julien Sopena[2] and Fabrice Legond-Aubry[2]

[1]Univ. degli Studi del Piemonte Orientale. Department of Computer Science.
Via Bellini 25G, 15100 Alessandria, Italy.
[2]LIP6 - Université de Paris 6
104, Avenue du President Kennedy, 75016 Paris, France.
souheib.baarir@mfn.unipmn.it, [julien.sopena,fabrice.legond-aubry]@lip6.fr

Abstract. In distributed environments, the shared resources access control by mutual exclusion paradigm is a recurrent key problem. To cope with the new constraints implied by recently developed large scale distributed systems like grids, mutual exclusion algorithms become more and more complex and thus much harder to prove and/or verify. In this article, we propose the formal modeling and the verification of a new generic hierarchical approach. This approach is based on the composition of classical already proof checked distributed algorithms. It overcomes some limitations of these classical algorithms by taking into account the network topology latencies and have a high scalability where centralized ones don't. We also have formalized the properties of the mutual exclusion paradigm in order to verify them against our solution. We prove that our compositional approach preserves theses properties under the assumption that all used plain algorithms assert them. This verification by formal method checkers was eased by the efficient use of already proved mutual exclusion algorithms and the reduction of state spaces by exploiting the symmetries.

keywords: distributed algorithm, composition, mutual exclusion, grid computing, colored Petri nets, model checking.

1 Introduction

By gathering geographically distributed resources, a Grid offers a single large-scale environment suitable for the execution of computational intensive applications. A Grid usually comprises of a large number of nodes grouped into clusters. Nodes within a cluster are often linked by local networks (LAN) while clusters are linked by a wide area network (WAN). Therefore, Grids present a hierarchy of communication delays: the cost of sending a message between nodes of different clusters is much higher than that of sending the same message between nodes within the same cluster.

Distributed or parallel applications that run on top of a Grid usually require that their processes get exclusive access to some shared resources (critical section). Thus, the performance of mutual exclusion algorithms is critical to Grid applications and it is the focus of this paper. A mutual exclusion algorithm ensures that exactly one process can execute the critical section at any given time (*safety* property) and that all critical section (CS) requests will eventually be satisfied (*liveness* property). We choose not to discuss the necessity, advantages or drawbacks of distributed versions of such algorithms. Readers can learn more informations about them in [7].

The contribution of this paper is two fold : the design of a generic hierarchical mutual exclusion composition approach which easily allows the combination of different *inter-cluster* and *intra-cluster* algorithms on the contrary to the previous approach and the verification of its correctness.

The remainder of this paper is organized as follows. Section 3 presents our composition approach and shows its advantages comparatively to existing works. In section 4, we describe the Petri net (P.N.) modelization of our approach followed by the expression of the properties we verify in section 5. Afterward, we present results of these properties verification on our proposed approach in section 7. The last section concludes our work and proposes interesting perspectives of research.

2 Related work

Several studies have proposed to adapt existing mutual exclusion algorithms to a hierarchical scheme. In Mueller [15], the author presents an extension to Naimi-Tréhel's algorithm, introducing the concept of priority. A token request is associated with a priority and the algorithm first satisfies the requests with higher priority. Bertier et al. [2] adopt a similar strategy based on the Naimi-Tréhel's algorithm which treats intra-cluster requests before inter-cluster ones.

Finally, several authors have proposed hierarchical approaches for combining different mutual exclusion algorithms. Housni et al. [8] and Chang et al. [3]'s mutual exclusion algorithms gather nodes into groups. Both articles basically consider hybrid approaches where the algorithm for intra-group requests is different from the inter-group one. In Housni et al. [8], sites with the same priority are gathered at the same group. Raymond's tree-based token algorithm [18] is used inside a group, while Ricart-Agrawala [19] diffusion-based algorithm is used between groups. Chang et al.'s [3] hybrid algorithm applies diffusion-based algorithms at both levels: Singhal's algorithm [20] locally, and Maekawa's algorithm [13] between groups. The former uses a dynamic information structure while the latter is based on a voting approach. Similarly, Omara et al. [17]'s solution is a hybrid of Maekawa's algorithm and Singhal's modified algorithm which provides fairness. In Madhuram et al. [12], the authors also present a two level algorithm where the centralized approach is used at lower level and Ricard-Agrawala at the higher level. Erciyes [6] proposes an approach close to ours based on a ring of clusters. Each node in the ring represents a cluster of nodes. The author then adapts Ricart-Agrawal to this architecture.

Our approach is close to these proposed solutions. However, we have found a more generic approach to achieve the scalability we need for large scale grid by finding a way to aggregate pre-existing algorithms and considering network latencies heterogeneity. It enables us to fit better the grid architecture and the application behavior. To do this, we have created glue code which coordinates two instance levels of plain mutual exclusion algorithms by just inserting well placed call traps in their inner code but without modifying their behavior. Practical results show significantly better performances [21] over classical distributed algorithms but no proof has been made to verify the correctness of the solution.

3 Our Composition Algorithm - An Informal Approach

Our approach consists in having a hierarchy of token-based mutual exclusion algorithms: a per cluster mutual exclusion algorithm that controls critical section requests from processes within the same cluster and a second algorithm that controls *inter-cluster* requests for the token. The former is called the *intra* algorithm while the latter is called the *inter* algorithm. An *intra* algorithm of a cluster runs independently from the other *intra* algorithms.

The application is composed of a set of processes which run on the nodes of the Grid. We consider one process per node and call it an *application* process. When an *application* process wants to access the shared resource, it calls the function *intra.CS_Request()*. It then executes its critical section. After executing it, the process calls the function *intra.CS_Release()* to release it. Both functions are provided by the *intra* token algorithm.

Within each cluster there is a special node, the *coordinator*. The *inter* algorithm runs on top of the *coordinators* allowing them to request the right of accessing the shared resource on behalf of *application* nodes of their respective cluster. *Coordinators* are in fact hybrid processes which participate in both the *inter* algorithm with the other *coordinators* and the *intra* algorithm with their cluster's *application* processes. However, even if the *intra* algorithm sees a *coordinator* as an *application* process, the *coordinator* does not take part in the application's execution *i.e.*, it never requests access to the CS for itself in the *intra* and *inter* layers but act as a **mandatory proxy for each layer**. As explained in the next sections, it forwards incoming *inter* requests and outgoing *intra* requests.

3.1 Coordinator algorithm

The key feature of our approach is that the two hierarchical algorithms are clearly separated since an *application* process gets access to the shared resource just by executing the *intra* algorithm of its cluster. Another important advantage is that the behavior of the chosen algorithms of both layers do not need to be modified. Hence, it is very simple to have different compositions of algorithms.

An *intra* algorithm controls an *intra* token while the *inter* algorithm controls an *inter* token. Thus, there is one *intra* token per cluster but a single *inter* token of which only the *coordinators* are aware. **Holding the *intra* token must be sufficient and necessary for an *application* process to enter the CS** since the local *intra* algorithm ensures that no other local *application* node of the cluster has the *intra* token. **But, considering the hierarchical composition of algorithms, our solution must then guarantee that no other *application* process of the other clusters is also in critical section when holding an *intra* token (per cluster *safety* property)**. In other words, the *safety* property of the *inter* algorithm must ensure that at any time only one cluster has the right of allowing its *application* processes to execute the CS. This property can be asserted by the possession of the *inter* token by a *coordinator*.

Similarly to a classical mutual exclusion algorithm, the *coordinator* calls the *inter.CS_Request()* and the *inter.CS_Release()* functions for respectively asking or releasing the *inter* token. However, when a *coordinator* is in critical section, it means that *application* processes of its cluster have the right of accessing the resource. The *inter* token is held by the *coordinator* of this cluster which is then considered to be in critical section by the other *coordinators*.

```
1  Coordinator Algorithm ()
2      intra.CS_Request()
3      /* Holds intra-token CS */
4      while TRUE do
5          if ¬ intra.PendingRequest() then
6              state ← OUT
7              Wait for intra.PendingRequest()
8          state ← WAIT_FOR_IN
9          inter.CS_Request()
10         /* Holds inter-token. CS */
11         intra.CS_Release()
12         if ¬ inter.PendingRequest() then
13             state ← IN
14             Wait for inter.PendingRequest()
15         state ← WAIT_FOR_OUT
16         intra.CS_Request()
17         /* Holds intra-token CS */
18         inter.CS_Release()

19 CS_Request ()
20     ...
21     mutexState ← $REQ$
22     Wait for Token
23     mutexState ← $CS$

24 CS_Release ()
25     ...
26     mutexState ← $NO\_REQ$

27 pendingRequest ()
28     return $\begin{cases} TRUE & \text{if } \exists \text{ pending request} \\ FALSE & \text{otherwise} \end{cases}$
```

Fig. 1. Coordinator Algorithm

Our composition solution does not require any change in the mutual exclusion algorithm. Providing such "plug in" feature is done by just inserting callbacks in the mutual exclusion implantation code. The algorithm themselves are not modified.

Only two trap callbacks are necessary: a *new request* trap and a *no more request* trap. The former, as its named suggest, must be invoked at each new token request processing while the latter must be invoked when there are no more pending request in the algorithm. These callbacks need no parameters and must be inserted in strategic code locations.

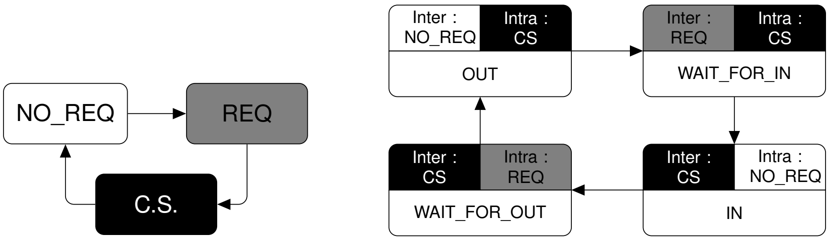
The guiding principle of our approach is described in the pseudo code of figure 1. Initially, every *coordinator* holds the *intra* token of its cluster and one cluster hold the *inter* token. When an *application* process wants to enter the critical section, it sends a request to its local *intra* algorithm by calling the *intra.CS_Request()* function. The *coordinator* of the cluster, which is the current holder of the *intra* token, will also receive such a request. However, before granting the *intra* token to the requesting *application* process, the *coordinator* must first acquire the *inter* token by calling the *inter.CS_Request()* function [line 9] of the *inter* algorithm. Therefore, upon receiving the *inter* token, the *coordinator* gives the *intra* token to the requesting *application* process by calling the *intraCS_Release()* function [line 11].

A *coordinator* which holds the *inter* token must also treat the *inter* token requests received from the *inter* algorithm. However, it can only grant the *inter* token to another *coordinator* if it holds its local *intra* token too. Having the latter ensures it that no *application* processes within its cluster is in the critical section. Thus, if the *coordinator* does not hold the *intra* token, it sends a request to its *intra* algorithm asking for it by calling the *intra.CS_Request()* function [line 16]. Upon obtaining the *intra* token, the *coordinator* can give the *inter* token to the requesting *coordinator* by calling the *inter.CS_Release()* function[line 18].

3.2 Coordinator automaton

In a classical mutual exclusion algorithm, a process can be in one of the three following states : requesting the critical section (REQ), not requesting it (NO_REQ), or in the critical section(CS), as shown in figure 2(a).

The behavior of a *coordinator* process can be summarized by a state automaton. A *coordinator* process is in one of the above three states in regards to both layer algorithms. **Therefore, in the automaton of figure 2(b),** *Intra* **and** *Inter* **refer to the** *coordinator* **state related to the** *intra* **algorithm and** *inter* **algorithm instance respectively.** Thus, a *coordinator* has new states in respect with the global state of the composition, which can be one of the following: OUT, IN, $WAIT\_FOR\_OUT$, $WAIT\_FOR\_IN$. These new states are a tuple composed of the states of each layer state.



(a) Classical mutual exclusion client automaton

(b) Coordinator automaton

Fig. 2. Coordinator and mutual exclusion client Automata

To ease the reader comprehension, we have had line references to the "**Coordinator algorithm**" pseudo-code in brackets and *inter* or *intra* layers state references of the automaton figure 2(b) in parenthesis. If the coordinator is in the state OUT, no local *application* processes of its cluster has requested the CS. Thus, it holds the *intra* token (*Intra* = CS)[line 2 or line 16] and does not hold the *inter* token (*Inter* = NO_REQ).

When the *coordinator* is in the state $WAIT\_FOR\_IN$, it means that there are one or more pending *intra* requests [line 7]. It still holds the local *intra* token (*Intra* = CS) but is waiting for the *inter* token (*Inter* = REQ)[line 9].

In the IN state, the coordinator holds the *inter* token (*Inter* = CS)[line 9] but has granted the *intra* algorithm token (*Intra* = NO_REQ)[line 11] to one of its *application* processes.

Finally, when the coordinator is in the state $WAIT\_FOR\_OUT$, it still holds the *inter* token (*Inter*= CS)[line 9] but it is requesting the *intra* token to the *intra* algorithm (*Intra* = REQ)[line 16] in order to be able to satisfy an *inter* algorithm pending request [line 14].

It is worth remarking that only one coordinator can be either in IN or in $WAIT\_FOR\_OUT$ state at any given time. All the other coordinators are either in state OUT or in state $WAIT\_FOR\_IN$.

4 Our Composition Algorithm - A Formal Model

High Level Petri Nets (*H.L.P.N.*) [9] formalism is an expressive model extending the representation of concurrency by Petri nets with a data management via the coloured domains and functions. It is well fitted for the representation of large distributed system like ours. Moreover, by use of Stochastic Well-Formed Petri nets (*S.W.N.*) [4], a particular category of H.L.P.N., we can check efficiently behavioral properties on the built representation. Thus, we have naturally choose this formalism over proof based methods. [1]

To obtain a good modelling, we have adopted an incremental and compositional methodology. We have isolated fundamental parts of our solution and defined Petri nets interfaces to bind them together. During the whole process, we have kept in mind the necessity to maintain the inherent symmetries of our approach. The preservation of behavioral symmetries is the key point to achieve our verification goals.

4.1 A basic Mutual Exclusion aware application modelization

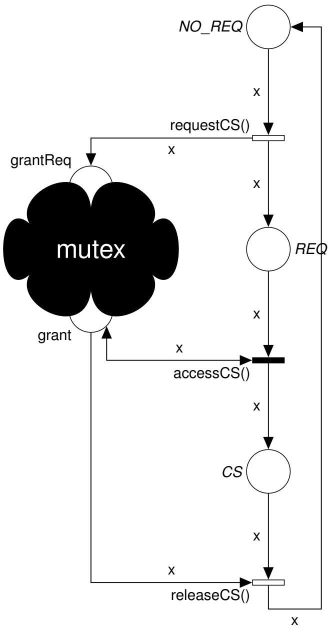


Fig. 3. Basic Mutual Exclusion modelling in *H.L.P.N.*

Distributed applications which use mutual exclusion can be summarized by a potentially infinite ordered succession of three specific states like those exposed in the section 3.2 and on the automaton of figure 2(a): *NO_REQ*, *REQ* and *CS*. These three states are represented by the three places at the right of the figure 3.

[1] It is worth noting that our models are described in the general framework of H.L.P.N., without taking into consideration the particular syntax of SWN. Actually, this simplifies considerably the modelling process without loss of generality.

Initially, the place NO_REQ contains a colored token per application process. A process do some local work during an undefined time and does not require an access to the exclusive resources. The need for a process to get the exclusive access is expressed by the firing of transition $requestCS()$. The processes identified by the colored token must then wait for the critical section (CS) granting authorization by the mutual exclusion algorithm. Upon clearance, the process token is then able to fire the transition $AccessCS()$ and will mark the place CS. The process can now execute its "critical section". As soon as it has finished (after an undefined time), it can get back to its local tasks by releasing the exclusive lock - *i.e.*, by firing the transition $releaseCS()$. Therefore the subnet composed of the places NO_REQ, REQ, CS and their adjacent transitions abstracts the behavior of our application processes.

The exclusive access to the place CS and the management of the request queue are ensured by a distributed mechanism: the mutual exclusion algorithm. This mechanism interacts with the application on every transitions. For now, we have no need to have a concrete modelling of such an algorithm, hence we abstract it by the use a *clouded* Petri net named "*mutex*" (see figure 3). At the border-side of the cloud, two places can be seen. The place $grantRequest$, when marked by a token x asserts the fact that the request for CS has been sent by the process x. The second is the place $grant$ which represents the mutual exclusion grant allowance for the process identified by the color of the token.

This model is in accordance with the classical A.P.I. of the mutual exclusion algorithms described in the pseudo-code of the figure 1: $CS\_Request()$ [line 19] and $CS\_Release()$ [line 24]. An application process use these two functions after a random elapsed time. This explains the temporisation of the corresponding transitions (white filled ▭). On the contrary, the firing of the transition $AccessCS()$ depends on the return of the "*wait for token*" synchronized blocking instruction call of the figure 1 pseudo-code [line 22]. So, the sojourn time in the place REQ is, deterministically, dependent of the availability of the grant token. This explains the immediate character of transition $AccessCS()$ (black filled ▬). As soon as the authorization is granted (the place $grant$ is marked by token x), the requesting process x enters the CS.

4.2 Our composition algorithm Petri net

Using the previous section, the modelling of our composition algorithm is much more simple. It can be seen as a synchronized use of two distinct instances of a mutual exclusion service: one at the *inter* level and one at the *intra* level. The subnet of the figure 4 models our composition approach. Since section 3 postulates the use of the same *intra* algorithm for each cluster, we have chosen to fold all the *intra* algorithm instances (*i.e.*, of every cluster) in one unique clouded subnet named "*intra*" at the right of the figure 4. To do so, the color token $\langle i, c\rangle$ identifies the process i of the cluster c. Note that the process color $\langle 0, c\rangle$ identify the coordinator. The c color permits the isolation of each local instances.

The subnet of the figure 4, composed of places NO_REQ, REQ and CS and its adjacent transitions abstracts the behavior of all the application processes inside each cluster. These are all the process $\langle i, c\rangle$ of each cluster c with $(i \neq 0)$ as explained later. From now on, we call it the "application subnet". Its places and transitions are not prefixed. Thus this subnet is nearly identical to the subnet of

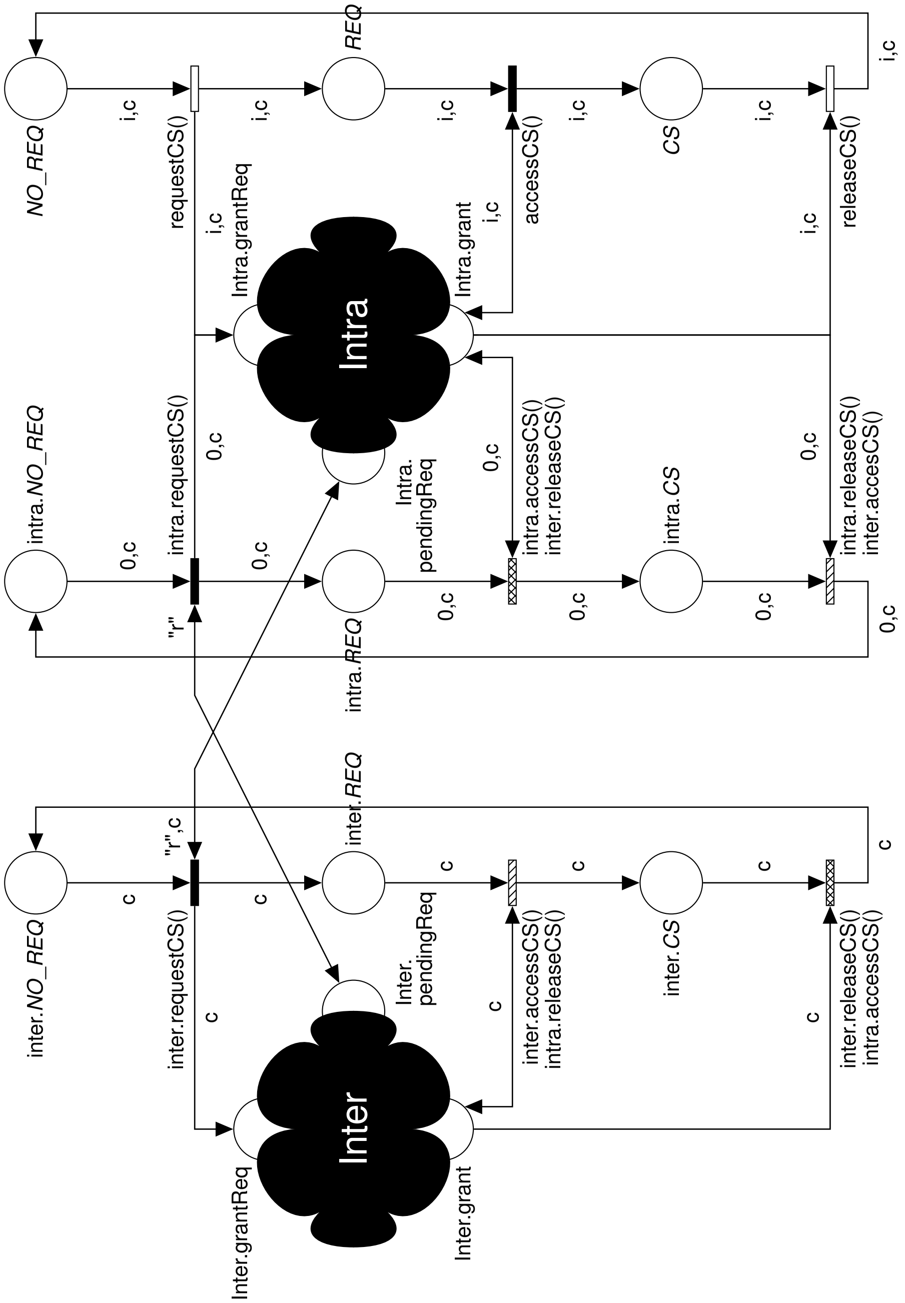


Fig. 4. *H.L.P.N.* of the composition algorithm

the figure 3 which illustrate the fact that the composition is nearly transparent from the application process point of view like described in the section 3.

However, inside each cluster, the coordinator processes which are identified by the color $\langle 0, c\rangle$ behave differently. In order to ease the interpretation of the global net, we unfold the application places of figure 3 for them. The places names will be prefixed by the "*intra*" mention. From now on, we call the subnet the "*intra* subnet". This subnet differs from figure 3 by its transitions. As explained in section 3, the coordinator does not act on its own initiative but just ensures the correctness of the solution (like uniqueness of the mutual exclusion grant token, . . .). The three transitions between the three places ”*intra.NO_REQ*”, ”*intra.REQ*” and ”*intra.CS*” are not temporized and controlled by our composition mechanism. That is why we need them all immediate transitions.

Finally, we set the "*inter* subnet" as the net composed of the prefixed *inter* places and its adjacent transitions. It defines the coordinator behavior with respect to the *inter* algorithm. There are only one coordinator by cluster on the whole grid so they only are identified by the color c in the *inter* subnet. And like the *intra* subnet, and for same reasons the three transitions between the three places ”*inter.NO_REQ*”, ”*inter.REQ*” and ”*inter.CS*” are immediate transitions.

Each coordinator has an *intra* behavior, based on the marking sequences of token $\langle 0, c\rangle$ the *intra* subnet and an *inter* behavior, based the marking sequences of token $\langle c\rangle$ the *inter* subnet. This abstraction enlighten the main idea of our solution, exposed via the automaton of figure 2(b): each state of the coordinator is a combination of an *inter* and *intra* local states.

To synchronize this dual behavior, we first split the *inter* and *intra* subnets into two main parts. The first concerns the *inter.requestCS*() and *intra.requestCS*() immediate transitions which trigger the sending of a request in its counterpart level. The second concerns the two immediate transitions called *inter.releaseCS*() / *intra.accessCS*() and *intra.releaseCS*() / *inter.accessCS*() which enforce the coordinated release of the CS and the grant allowance of each level. **These two transitions have been split to ease the reading of the model but they are filled with the same patterns to clearly identify them.**

A coordinator request sending can be viewed as a *forward* from one level to the other. So the transition firing is enforced by the reception of an *inter* or *intra* request. We need to materialize this information inside the *inter* and *intra* algorithms for the coordinators to exploit them. Thus, in the figure 1 pseudo-code, we need to add a *pendingRequest*() function [line 27] to the standard A.P.I.. To abstract this reification we have added a new state called *pendingReq* at the *inter* and *intra* clouded P.N. border-side. The marking of the place *inter.pendingReq* (resp. *intra.pendingReq*) represents the registration of a request coming from the *inter* (resp. *intra*) layer.

The *inter* (resp. *intra*) critical section coordinated release is enforced by the real access (and thus the grant authorization) to the *intra* (resp. *inter*) section. Abstracting this behavior must be done by a cross-synchronization between the *accessCS*() action of one layer with the *releaseCS*() on the other - like on the figure 2(b). On figure 4, the *inter.accessCS*() / *intra.releaseCS*() [lines 9 and 11] and *intra.releaseCS*() / *inter.accessCS*() are the same immediate transition (▨) and represents this desired synchronization. So does the split transition *intra.accessCS*() / *inter.releaseCS*() (▩) [lines 16 and 18].

To finalize our model, we specify the initial global marking of the system. To achieve this, we define the following sets:

C **:** the finite set of all clusters
$C_{c'} = \{c \in C | c \neq c'\}$**:** the finite set of all clusters minus the cluster element c'.
A_c **:** the finite set of all processes of a single cluster c.
$A^*_c = \{i \in A_c | i \neq 0\}$ **:** the finite set of all **application** processes of a single cluster c. The application processes are the set of all processes i of the cluster c minus the coordinator process (with index $i = 0$).

Under the hypothesis where $M(p)$ represents the marking of the place p, the initialisation is performed by the following markings:

- all the application nodes are not requesting the CS, thus are in the NO_REQ state.
$$M(NO\_REQ) = \sum_{c \in C} \sum_{i \in A^*_c} \langle i, c \rangle$$
- the coordinator c' is in the CS state) w.r.t. the *inter* algorithm but in the NO_REQ state w.r.t. the intra algorithm.
$$M(intra.NO\_REQ) = \langle 0, c' \rangle \quad \text{and} \quad M(inter.CS) = M(inter.grant) = \langle c' \rangle$$
- all other coordinators are in CS state w.r.t. the *intra* algorithm and are in the NO_REQ state w.r.t. the *inter* algorithm.
$$M(inter.NO\_REQ) = \sum_{c \in C_{c'}} \langle c \rangle \quad \text{and} \quad M(intra.CS) = M(intra.grant) = \sum_{c \in C_{c'}} \langle 0, c' \rangle$$

5 Fundamental properties

The mutual exclusion paradigm was first introduced and informally defined by Dijkstra in 1965 [5]. This article has defined the bases of the mutual exclusion problem and was successively refined into more formal definitions [11]. Defining mutual exclusion is to define a set of properties that must be asserted by all algorithms of this paradigm. These properties are:

Well-formedness: all the processes must respect the classical automaton of mutual exclusion, as described in figure 2(a).
Mutual Exclusion: at any time, there is at most one process in the CS state (figure 2(a))
Progress: if there is at least one process in the REQ state and there is no process in the CS state, then eventually one process will enter in the REQ state.

Following the Lamport [10] taxonomy, the first two properties can be classified in the safety class properties. The last one can be put in the liveness class properties. However, the *Progress* liveness property does not guarantee for a process to access the CS. Rather, it is a global notion of liveness. So to avoid any starvation for a particular process, a mutual exclusion algorithm must verify a complementary property:

Weak fairness: if one process is in the REQ state and if the mutual exclusion section execution time is finite, then the process will eventually access to the CS state.

This weak fairness property implies the progress property because the individual liveness implies the system wide liveness. But as many applications can not afford to rely on the progress alone, many articles do not even consider progress and instead use the weak fairness property. In the remaining of this paper, we consider these two properties distinctively and we explicit which one is used.

5.1 Formal expression of properties

The aforementioned properties can all be expressed using the Linear Temporal Logic (LTL). We begin by defining some atomic propositions that will help us to translate mutual exclusion properties into LTL.

- P_1 : the process i of the cluster c does not require the CS nor is in CS ($M(NO\_REQ) \geqslant \langle i, c\rangle$).
- P_2 : the process i of cluster c requests an access to the CS ($M(REQ) \geqslant \langle i, c\rangle$).
- P_3 : the process i of the cluster c is in CS ($M(CS) \geqslant \langle i, c\rangle$).
- P_4 : the process i of the cluster c is NOT in CS ($M(CS) < \langle i, c\rangle$).
- P_5 : the number of application processes in CS is less or equal than 1 ($\#(CS) \leqslant 1$).
- P_6 : there is no application process in CS (no one is in place CS). ($\#(CS) = 0$).
- P_7 : there is a exactly one application process in CS ($\#(CS) = 1$).
- P_8 : there is at least one application process which request an access to the CS ($\#(REQ) \geqslant 1$).

Then the properties can be written down as follows:

Well-formedness: if a process marks the place NO_REQ (resp. REQ, CS), it will not be able to mark the place CS (resp. NO_REQ, REQ) without having previously marked the place REQ (resp. CS, NO_REQ).

$F_1 : G(P_1 \Rightarrow F(!P_3\ U\ P_2)) \wedge G(P_2 \Rightarrow F(!P_1\ U\ P_3)) \wedge G(P_3 \Rightarrow F(!P_2\ U\ P_1))$

Mutual Exclusion: there is always at most one application process in the CS state.

$$F_2 : G(P_5)$$

Progression: always, if there is at least one application process requesting the CS (*i.e.*, a token $\langle i, c\rangle$ marks the place REQ) and if there is no process in CS, then an application process will be able to access the CS (*i.e.*, it will mark the place CS).

$$F_3 : G((P_8 \wedge P_6) \Rightarrow F(P_7))$$

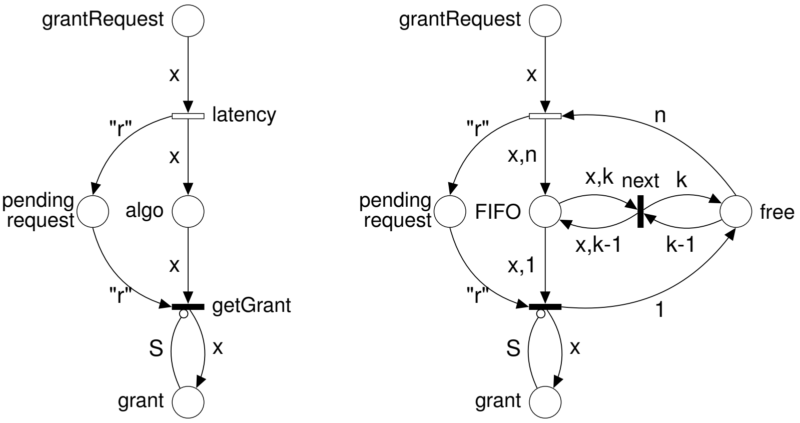
Weak fairness: one application node will always be able to access the CS after having requesting it.

$$F_4 : G(P_2 \Rightarrow F(P_3))$$

6 Simplified models for mutual exclusion algorithms

To check the previously defined properties on our composed mutual exclusion algorithm we need to instantiate the *inter* and *intra* clouded nets. Two methods would have been possible. The first one consists in replacing each clouded net by a *H.L.P.N.* reflecting the exact behavior of some well known mutual exclusion algorithms like [22], [16] or [14]. However, this level of details is only useful for quantitative studies and to evaluate the effect of each algorithm on our composition for, for instance, the "mean delay transmission time" or "the mean number of exchanged requests", etc. The second method consists in simply modelling the properties they assert. The aim is then to check if our composition approach upholds the properties of the algorithms it uses.

In this paper, we have chosen the second approach which enables us a preliminary qualitative study of our solution. It is a necessary step prior any real quantitative study. Thus, we propose two H.L.P.N. models which verify the properties described in section 5. The figure 5(a) H.L.P.N. abstracts the *validity, mutual exclusion* and *progress* properties whereas figure 5(b) H.L.P.N. abstracts the *validity, mutual exclusion, progress* and *weak fairness.*



(a) Abstraction not asserting Weak Fairness

(b) Abstraction asserting Weak Fairness

Fig. 5. Mutual Exclusion algorithm abstraction nets.

Consider figure 5(a), we observe the presence of the places *grantRequest*, *pendingReq* and *grant* at the border-side of figure 4. We also have a place *algo* which materializes the request treatment. The transition *latency* stands for the request reception event. Trivially, everyone can check we do not consider the request transmission method: it can be a simple message emission like in Suzuki-Kasami [22] or a sequence of them like in Martin algorithm [14]. As the network travelling time and the registering treatment time are undetermined, the transition is temporized (white filled ▭). The CS access is modeled by the *getGrant* transition. The exclusive access is ensured by the inhibitor arc on the place *grant*. The *progress* property on the registered requests (place *algo*) is provided by the immediate transition *getGrant* (black filled (▬) .

To continue the description of the figure 5(a), lets notice that places *algo* and *pendingRequest* do not have the same color domains. This is due to the fact that our composition algorithm only need to know if there is any request that must be treated but do not need to know which is the requesting process. So, when the transition *latency* is fired by a token $\langle x \rangle$, the *pendingReq* place is, at the same time, marked by a constant "*r*"(to notice the reception of the request by the algorithm). So, the requesting process identity remains unknown to the coordinator.

To conclude, this H.L.P.N. does not guarantee *weak fairness*: some tokens in place *algo* can potentially never pass through the transition *getGrant*. They can be perpetually overtaken by new incoming requests.

Now, the figure 5(b) enhances the previous H.L.P.N. by substituting to the place *algo* a model of a fair request queue (in fact, it is a simple FIFO queue of size n). The queue is modelled by two places. The first one is the place $FIFO$ marked with colored token $\langle x, k\rangle$. When a request of process x, comes in, it is associated to a position k starting with the last position (index n). The second one is the place $free$ which is initially marked by all available positions - *i.e.*, all the k colors. When a request marks the place $FIFO$, its position will progress by firing the immediate transition *next*. But a request x with position k can only fire the transition $free$ if the $k-1$ position is available (*i.e.*, only if its predecessor was able to progress). Thus all request are treated in their arrival order and the *weak fairness* property is asserted for all the registered requests - *i.e.*, all requests that have fired the transition *latency*. However, asserting this property for all sent requests (*i.e.*, for every token marking the *grantRequest*) is another problem. It requires the modelling of an additional hypothesis. Actually, all mutual exclusion algorithms make this following minimum hypothesis about their communication channels: we never lost the same message twice. So to say, a message sent an infinity number of times will be received an infinity number of times. This property is called the *"fair lossy channel"* property. The integration of this hypothesis can be done in two ways: the first one is to make the transition *latency* firing “fair”. This materializes the fact that each request will be registered by the mutual exclusion algorithm. For each -infinite- execution of our model, if the transition *latency* is firable then it will eventually be fired. The second way is to take this constraint directly in the properties. We modify the properties in order to exclude all the scenarios where at least one specific x marking the place *grantRequest* do not fire the *latency* transition on the whole execution.

The second solution has been chosen because H.L.P.N in their classical definition do not enable us to set a transition as “fair‘”. Thus, we have rewritten the property F_4 and we use the following atomic propositions to do it:

- P_9 : an *intra* request of the process i' of the cluster c' has not been treated - *i.e.*, it was sent but not registered by the used mutual exclusion algorithm $(M(intra.grantRequest) \geqslant \langle i', c'\rangle)$.
- P_{10} : an *inter* request of the cluster c'' coordinator has not been treated - *i.e.*, it was sent but not registered by the used mutual exclusion algorithm $(M(inter.grantRequest) \geqslant \langle c''\rangle)$.

Hence, the *weak fairness* property must be modified as follows:

weak fairness: always, if a process i of the cluster c request for the CS then, either in the future, it will have the CS, or at least one message for the process i' of the cluster c' will be treated, or at least one message of the coordinator of the cluster c'' will be treated.

$$F_4 = G(P_2 \Rightarrow (F(P_3) \vee FG(P_9) \vee FG(P_{10}))$$

7 Model checking

The classical method to verify a model (*i.e.*, *model-checking* [23]) against LTL properties relies on automata theory. Within this approach, all possible executions of the studied application are produced and synchronized with the automaton representing the executions *invalidating* the desired property. If the resulting automaton is “not empty” then the property is not satisfied by the model. Here “not empty” means that the language recognized by the automaton is not reduced to the empty word.

The main problem of this approach is the excessive size of the generated automata. Actually, This size can be exponentially greater than the syntactic description of the model and the property (the well-known state space explosion problem). The explosion is essentially due to the concurrency of the system actions and thus the synchronisation of its elements. Many approaches were developed to overcome this problem. Their aims either are to drastically reduce the representation of the generated automata or to substitute context-equivalent smaller automata. One of these last such solutions is based on the observation that concurrent systems are composed of identical behaviors (up to a permutation). The factorisation of the representation of such similar behaviors leads to the construction of smaller automata which can be efficiently used for model checking [1].

Our composition approach is highly symmetric. In fact, we have identified and used symmetries at all levels: the application process of the same cluster behave the same and so do the coordinators process. Hence, we have kept and used them in all our modeling process. Moreover, by use of the rigorous syntax of SWN these symmetries are efficiently represented and exploited for an *automatic* construction of a reduced automaton representing the system executions [4, 1]. These ends up by the verification of our properties.

The tool we used to generate the reduced automaton of our model is the well know and widely used GreatSPN[2]. It was connected to the Spot[3] model-checker tool. The verification is done in two steps. Firstly, we verified the mutual exclusion algorithm models (figure 5) by plugging them into the (abstract) model of the application process (figure 3). Secondly we have plugged the model of mutual exclusion algorithm (figure 5) in the abstract model of figure 4. Trivially, the first part was checked and the properties F_1 to F_3 were verified on the model of figure 5(a) and the properties F_1 to F_4 were verified on the model of figure 5(b).

For the second part, and the most important, the results show that all properties are preserved: when the used algorithm verify *validity, mutual exclusion* and *progress* for the *intra* and *inter* levels our solution validate the same properties. When the used *intra* and *inter* algorithms verify the *validity, mutual exclusion, progress* and *weak fairness* properties (F_1 to F_4) our algorithm does the same way whichever the topology we choose.

To give an idea on the complexity of the model-checking accordingly to a chosen deployment topology, we highlight in table 1 some of the obtained results. Here we represent the number of visited states for the verification of each of the described properties, when using the model of 5(b) to instantiate the composition approach model.

Six topologies noted ”xc./ya.” are reported into it. In our notation, x is the number of clusters (that is why it is postfixed by c) and y is the number of application processes by cluster (that is why it is postfixed by a). So, we have checked six topologies: three with 6 application processes gathered into 1, 2 and 3 clusters and

[2] http://www.di.unito.it/ greatspn/

[3] http://spot.lip6.fr/

Topo.	6 process			8 process		
Propr.	6 a.	2 c. 3 a.	3 c. 2 a.	8 a.	2 c. 4 a.	4 c. 2 a.
F_1	1438	70823	145455	2888	619362	1793654
F_2	218	9988	20205	391	74817	212666
F_3	318	14548	30662	569	108295	320577
F_4	785	36844	76018	1716	345375	708019

Table 1. Model-checking over different topologies

three with 8 application processes gathered into 1, 2 and 4 clusters. The topology noted "ya." is the plain algorithm used as complexity reference which is low comparing to our composition. Increasing the number of clusters generate more states because of the synchronizations implied by our composition.

8 Conclusion

This paper exposes a new algorithm to easily compose existing mutual exclusion algorithms in order to achieve better scalability on grids. This solution enable us to optimize the grant authorization time without a lost of the basic mutual exclusion properties. It is also totally transparent for applications.

To check the consistency of our solution, we have isolated mutual exclusion algorithm common A.P.I. We have modelled this generic A.P.I. into H.L.P.N. We take good advantages of this defined interface to compositionally put together our modelling. Based on these A.P.I., we were able to plug in mutual exclusion algorithm abstractions that assert the classical mutual exclusion paradigm properties. This simplification, sufficient for this first qualitative study, make possible to model-check our composition algorithm against the same properties. Concerning these properties, we have done their LTL conversion and integrated an underlying crucial hypothesis called *"fair lossy channel"* required by almost all mutual exclusion algorithms.

During the whole modelling process and verification, we always kept in mind the inner symmetries of our solution. After exhibiting them in our algorithm, we has exploited them to best model our solution and maximize the simplification the LTL properties. At last, the conservation of these symmetries was exploited in the model-checking by using specific algorithms.

This study has numerous research perspectives. The fine P.N. modelling of existing -classical- mutual exclusion algorithms like Suzuki and Kasami [22], Naimi-Tréhel [16] or Martin[14] could lead to numerical quantitative study of the influence of our solution with respect to the application processes. We will be able to calculate performance indices accordingly to the composed plain algorithms.

References

1. S. Baarir, S. Haddad, and J-M. Ilié. Exploiting Partial Symmetries in Well-formed nets for the Reachability and the Linear Time Model Checking Problems. In *Proceeding of IFAC Workshop on Discrete Event Systems, part of 7th CAAP*, Reims - France, 2004. Springer Verlag.
2. M. Bertier, L. Arantes, and P. Sens. Distributed mutual exclusion algorithms for grid applications: A hierarchical approach. *Journal of Parallel and Distributed Computing*, 66:128–144, 2006.

3. I. Chang, M. Singhal, and M. Liu. A hybrid approach to mutual exclusion for distributed system. In *IEEE International Computer Software and Applications Conference*, pages 289–294, 1990.
4. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed coloured nets for symmetric modelling applications. *IEEE Transactions on Computers*, 42(11):1343–1360, nov 1993.
5. Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Comm. ACM*, 8(9):569, 1965.
6. K. Erciyes. Distributed mutual exclusion algorithms on a ring of clusters. In *Intern. Conf. on Computational Science and Its Applications*, volume 3045 of *LNCS*, pages 518–527, 2004.
7. S. Fu, N. Tzeng, and Z. Li. Empirical evaluation of distributed mutual exclusion algorithms. pages 255–259.
8. A. Housni and M. Trhel. Distributed mutual exclusion by groups based on token and permission. In *International Conference on Computational Science and Its Applications*, pages 26–29, June 2001.
9. K. Jensen. High-level petri nets. In *Proceedings of the 3rd European Workshop on Application and Theory of Petri Nets*, Varenna - Italy, 1982.
10. Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.
11. Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
12. Madhuram and Kumar. A hybrid approach for mutual exclusion in distributed computing systems. In *IEEE Symposium on Parallel and Distributed Processing*, 1994.
13. M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM-Transactions on Computer Systems*, 3(2):145–159, May 1985.
14. A. J. Martin. Distributed mutual exclusion on a ring of processes. *Sci. Comput. Program.*, 5(3):265–276, 1985.
15. F. Mueller. Prioritized token-based mutual exclusion for distributed systems. In *International Parallel Processing Symposium*, pages 791–795, March 1998.
16. M. Naimi and M. Trehel. An improvement of the log(n) distributed algorithm for mutual exclusion. In *IEEE Intern. Conf. on Distributed Computing Systems*, pages 371–377, 1987.
17. F. Omara and M. Nabil. A new hybrid algorithm for the mutual exclusion problem in the distributed systems. *International Journal of Intelligent Computing and Information Sciences*, 2(2):94–105, 2002.
18. K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.
19. G. Ricart and A. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24, 1981.
20. M. Singhal. A dynamic information structure for mutual exclusion algorithm for distributed systems. *IEEE Trans. on Parallel and Distributed Systems*, 3(1):121–125, 1992.
21. J. Sopena, F. Legond-Aubry, L. Arantes, , and P. Sens. A composition approach to mutual exclusion algorithms for grid applications. In *Proc. of the International Conference on Parallel Processing*, page 65, 2007.
22. I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 3(4):344–349, 1985.
23. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham M. Birtwistle, editors, *Proceedings of the 8th Banff Higher Order Workshop*, volume 1043 of *LNCS*, pages 238–266. Springer-Verlag, 1996.