Distributed Verification of Modular Systems

M.C. Boukala[1] and L. Petrucci[2]

[1] LSI, Computer Science department, USTHB
BP 32 El-Alia Algiers, ALGERIA
boukala@lsi-usthb.dz

[2] LIPN, CNRS UMR 7030, Université Paris XIII
99, avenue Jean-Baptiste Clément
F-93430 Villetaneuse, FRANCE
Laure.Petrucci@lipn.univ-paris13.fr

Abstract. The use of distributed or parallel processing gained interest in the recent years to fight the state space explosion problem. Many industrial systems are described with large models, and the state space being even larger, it does not fit completely into the memory of a single computer.

To avoid the high space requirement, several reduction techniques have been proposed: modular verification, partial order reductions, symmetries, using symbolic or compact representations like BDDs.

Another way to alleviate the state space explosion problem is to use modular analysis, which takes advantage of the modular structure of a system specification, particularly for systems where the modules exhibit strong cohesion and weak coupling.

In this paper, we propose to combine distributed processing and modular analysis to perform verification of basic behavioural properties such as reachability, deadlock states, liveness, and home states and their distributed analysis for modular systems. Each module is assigned to a process which explores independently the internal activity of the module, allowing a significant reduction in the size of the state space rather than in an interleaved fashion.

Keywords: Modular systems, distributed verification, modular analysis, Petri nets.

1 Introduction

Systems developed nowadays are both more and more complex and critical. When addressing the design of such systems, it is necessary to ensure reliability, by verifying the system properties. The main approach to verification consists in the generation and analysis of the state space. Some properties such as reachability or deadlocks can be verified on-the-fly, and only require information on states reachable from the initial marking. On the contrary, liveness and home states are elaborate properties which require a full generation of the state space including not only nodes but also arcs. Even for small models, the size of the state space may be too huge to fit in the memory of a single computer.

To cope with the state space explosion problem, several reduction techniques have been proposed: on-the-fly verification checks properties during the state space construction; sweep-line construction [Sch04] considers a progress measure, and discards states that will not be encountered further in the construction; modular verification [LP04]; partial order reductions [Val92,NG97]; symmetries [AHI98,CFJ93]; using symbolic or compact representations like BDDs and Kronecker algebra.

Recently, several studies addressed distributed verification, many of them focussing on distributed LTL model-checking [BBS01,BO03,LS99,BP07]. These works are mainly based on the partionning of the state space.

The performances of distributed verification depends on several criteria, e.g. load balancing of the partitioned state space, but also, more importantly, on a good partitioning. Therefore, choosing an adequate hash function to assign nodes to processors is important.

In [LP04] the modular analysis approach examines in isolation the local behaviour of each subsystem, and then separately considers the synchronisation between the subsystems. In this fashion, exploring the many possible interleavings of activity of the subsystems is avoided, thus reducing the state space.

In this work, we propose to combine modular analysis and distributed processing to improve consequently the systems verification. We focus on checking usual properties such as reachability, liveness and home states.

The paper is organised as follows. We assume the reader is familiar with basic Petri nets notions. Hence, we recall in Section 2 only the modular concepts, i.e. Modular Petri nets and Modular State Spaces. In section 3, we introduce algorithms based on distributed modular construction of the state space. In section 4 distributed verification of various properties is presented. These algorithms have been implemented in a prototype tool and experimental results are presented in section 5. Finally, section 6 concludes the paper.

2 Modular Petri Nets

In this paper, we consider only modules synchronised through shared transitions as in [LP04].

2.1 Definition of Modular Petri Nets

Definition 1 (Modular Petri net). *A* modular Petri net *is a pair $MN = (S, TF)$, satisfying:*

1. *S is a finite set of* modules *such that:*
   - *Each module, $s \in S$, is a Petri net: $s = (P_s, T_s, W_s, M_{0_s})$.*
   - *The sets of nodes corresponding to different modules are pair-wise disjoint:*
     $\forall s_1, s_2 \in S : [s_1 \neq s_2 \Rightarrow (P_{s_1} \cup T_{s_1}) \cap (P_{s_2} \cup T_{s_2}) = \emptyset]$.

– *$P = \bigcup\limits_{s \in S} P_s$ and $T = \bigcup\limits_{s \in S} T_s$ are the sets of all places and all transitions of all modules.*

2. $TF \subseteq 2^T$ is a finite set of non-empty transition fusion sets*.*

In the following, TF also denotes $\bigcup_{tf \in TF} tf$. We now introduce transition groups.

Definition 2 (transition group). *A* transition group *$tg \subseteq T$ consists of either a single non-fused transition $t \in T \setminus TF$ or all members of a transition fusion set $tf \in TF$.*
The set of transition groups is denoted by TG.

A transition can be a member of several transition groups as it can be synchronised with different transitions (a sub-action of several more complex actions). Hence, a transition group corresponds to a synchronised action. Note that all transition groups have at least one element.

Next, we extend the arc weight function W to transition groups, i.e. $\forall p \in P, \forall tg \in TG$:

$$W(p, tg) = \sum_{t \in tg} W(p, t), \quad W(tg, p) = \sum_{t \in tg} W(t, p).$$

Markings of modular Petri nets are defined as markings of Petri nets, over the set P of all places of all modules. The restriction of a marking M to a module s is denoted by M_s. The enabling and occurrence rules of a modular Petri net can now be expressed.

Definition 3 (Transition group enabledness). *A transition group tg is* enabled *in a marking M, denoted by $M[tg\rangle$, iff:*

$$\forall p \in P : W(p, tg) \leqslant M(p)$$

When a transition group tg is enabled in a marking M_1, it may occur, changing the marking M_1 to another marking M_2, defined by:

$$\forall p \in P : M_2(p) = (M_1(p) - W(p, tg)) + W(tg, p).$$

Figure 1 depicts a modular Petri net consisting of three modules A, B and C. Modules A and B both contain transitions labelled F_1 and F_3, while modules B and C both contain transition F_2. These matched transitions are assumed to form three transitions fusion sets.

Example: The (full) state space for the modular Petri net of figure 1 is shown in figure 2. Note that the initial state is shown as $A_1B_1C_1$, thus indicating that place A_1 is marked with a token in module A, place B_1 is marked with a token in module B, and place C_1 is marked with a token in module C. In this initial state, only transition F_1 is enabled, its occurrence leading to state $A_2B_2C_1$.

When considering the modular state space, as well as checking properties of the system, we will use Strongly Connected Components. The set of all strongly connected components is denoted by SCC. For a node v and a component $c \in SCC$ we use $v \in c$ to denote that v is one of the nodes in c. A similar notation is used for arcs. We use v^c to denote the component to which v belongs.

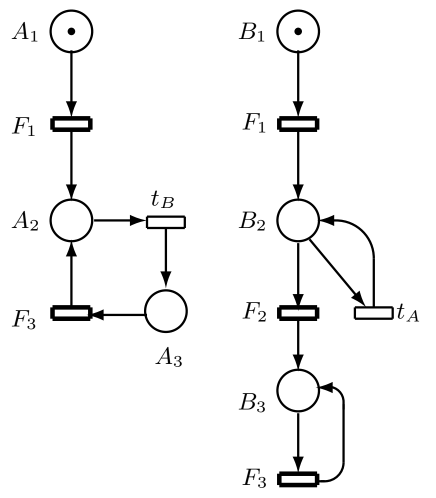

Fig. 1: A modular Petri net with 3 modules

2.2 Modular State Spaces

In the definition of modular state spaces, we denote the set of states reachable from M by occurrences of local (non-fused) transitions only, in all the individual modules, by $[[M\rangle$.

The notation with a subscript s means the restriction to module s, e.g. $[M\rangle_s$ is the set of all nodes reachable from the global marking M by occurrences of transitions in module s only.

We use $M[[\sigma\rangle\rangle M'$ to denote that M' is reachable from M by a sequence $\sigma \in (T \setminus TF)^*TF$ of internal transitions followed by a fused transition. In the definition of modular state spaces we need a compact notation to capture the states reachable from M in all the individual modules. It turns out that we can use a product of SCCs of the individual modules to express this representative node: for any reachable marking M, we use M^c to denote the product (or tuple) of Strongly Connected Components ($SCCs$) M_s^c of the individual modules:

$$\forall M \in [M_0\rangle : M^c = \prod_{s \in S} M_s^c.$$

The definition of a modular state space consists of two parts: the state spaces of the individual modules and the synchronisation graph.

Definition 4 (Modular state space). *Let $MN = (S, TF)$ be a modular Petri net with the initial marking M_0. The* modular state space *of MN is a pair $MSS = ((SS_s)_{s\in S}, SG)$, where:*

1. *$SS_s = (V_s, A_s)$ is the* local state space *of module s:*
   (a) $V_s = \bigcup_{v \in V_{SG}} [v\rangle_s$

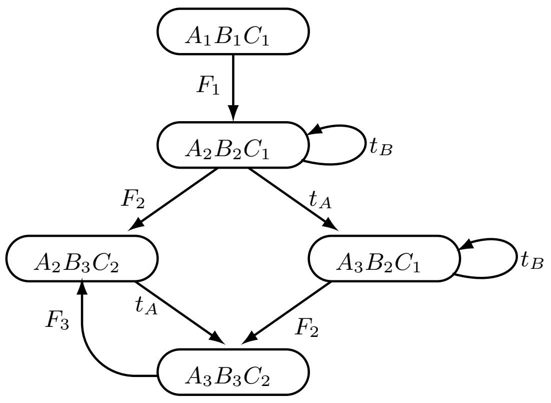

Fig. 2: The state space of the modular net of figure 1

   (b) $A_s = \{(M_1, t, M_2) \in V_s \times (T \setminus TF)_s \times V_s \mid M_1[t\rangle M_2\}$
2. $SG = (V_{SG}, A_{SG})$ *is the* synchronisation graph *of MN:*
   (a) $V_{SG} = [[M_0\rangle\rangle^c \cup M_0^c$
   (b) $A_{SG} = \{(M_1^c, (M_1^{'c}, tf), M_2^c) \in V_{SG} \times ([M_0\rangle^c \times TF) \times V_{SG} \mid M_1^{'} \in [[M_1\rangle \wedge M_1^{'}[tf\rangle M_2\}$

A detailed explanation of the definition is given in [LP04]:
(1) The definition of the state space graphs of the modules is a generalization of the usual definition of state spaces.

(1a) The set of nodes of the state space graph of a module contains all states locally reachable from any node of the synchronisation graph.

(1b) Likewise the arcs of the state space graph of a module correspond to all enabled internal transitions of the module.
(2) Each node of the synchronisation graph is labelled by a M^c and is a representative for all the nodes reachable from M by occurrences of local transitions only, i.e. $[[M\rangle$. The synchronisation graph contains the information on the nodes reachable by occurrences of fused transitions.

(2a) The nodes of the synchronisation graph represent all markings reachable from another marking by a sequence of internal transitions followed by a fused transition. The initial node is also represented.

(2b) The arcs of the synchronisation graph represent all occurrences of fused transitions.

The state space graphs of the modules contain only local information, i.e. the markings of the module and the arcs corresponding to local transitions but not the arcs corresponding to fused transitions. All the information concerning these is stored in the synchronisation graph.

The nodes of the synchronisation graph represent all markings reachable from another marking by a sequence of internal transitions followed by a fused transition. The initial node is also represented.

The arcs of the synchronisation graph represent all occurrences of fused transitions. Each arc is labelled by the corresponding fired transition and by the SCCs of the markings which enabled this transition. But only the SCCs of the participating modules do appear.

Example: The modular state space for the modular Petri net of figure 1 is shown in figure 3. Note that there is a local state space for each module, as well as a synchronisation graph which captures the occurrence of fused transitions. We do not distinguish between nodes and SCCs since, in this case, all SCCs consist of a single node (which is seldom the case in practice).

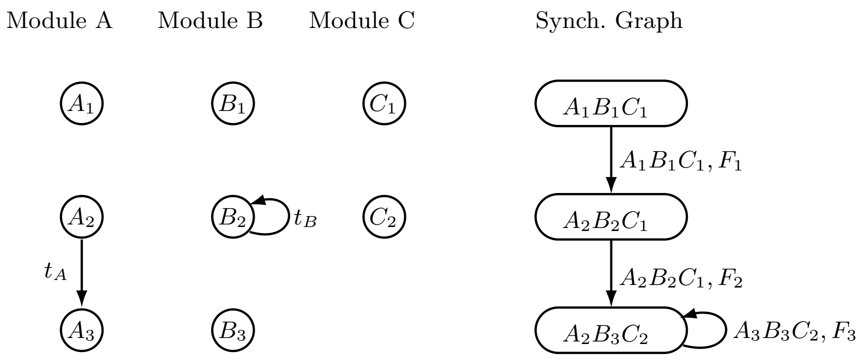

Fig. 3: The modular state space of the net in figure 1

Although sketches of algorithms were introduced in [LP04], we give here the description of the distributed algorithms for the construction of the state space and for the verification of the reachability, dead markings, liveness, and home states properties.

3 Distributed Construction of the Modular State Space

Several works have developed distributed tools generating and exploring the state space on a cluster of workstations. Partitioning the set of states over the different stations is done using a hash function [GMS01,KP04,AAC87,LP04]. This approach can handle larger state spaces but still remains limited.

Here, we consider a different approach based on modularity to achieve a distributed construction of the state space and the verification of various properties.

In this approach we also used two types of processes. They consist in a *coordinator process* and *N worker processes*, one worker process for each module, as in [KP04].

The coordinator constructs the synchronisation graph, coordinates the different worker processes and determines the termination of the modular state space construction, while every worker generates the local state space of the module it is assigned by firing only internal transitions.

The coordinator process starts by sending the initial marking M_{0_s} restricted to module s to each worker process, and waits for receiving the states enabling fused transitions from the workers processes.

Algorithm 1: Internal State Space Generation()

```
begin
    /* Process states in $Waiting$ */
    while $\neg EndOfGeneration$ do
        while $Waiting \neq \emptyset$ do
            Choose $M \in Waiting_s$
            forall the $t \notin TF$ $s.t.$ $M[t\rangle M'$ do
                /* $t$ is an enabled internal transition */
                Add $(M')$
                Arc $(M, t, M')$
                UpdateSCC ()
            forall the $tf \in TF$ $s.t.$ $M[tf\rangle$ do
                /* $M$ enables a fused transition $tf$ */
                $TFWaiting \leftarrow TFWaiting \cup \{(M, tf)\}$;
            $Waiting_s \leftarrow Waiting_s \setminus \{M\}$
        if $TFWaiting = \emptyset$ then
            $EndOfGeneration = true$
        else
            while $TFWaiting \neq \emptyset$ do
                Choose $(M, tf) \in TFWaiting$ s.t. $M[tf\rangle M'$
                $Waiting_s \leftarrow Waiting_s \cup \{M'\}$
                $TW.label = M^c$
                $TW.AncSCC = AncSCC(M)$
                $TW.tr = tf$
                $TW.SuccSCC = M'^c$
                Send ($Sender = s$, $Dest = Coordinator$, $Type =$ SynchTrans,
                        $Content = TW$)
    Send ($Sender = s$, $Dest = Coordinator$, $Type =$ END_GENERATION)
end
```

When a worker process s receives the initial marking, it adds it to the sets V_s and $Waiting_s$, that correspond respectively to the set of local markings and the set of markings which are visited but not explored yet. The worker process then invokes algorithm 1 to generate the internal state space of the module it is assigned. Function `Add` (M) adds the marking M to both sets V_s and $Waiting_s$ if it is not already in V_s.

The workers send enabled fused transitions to the coordinator process. Actually, the message sent to the coordinator not only contains the fused transition *tf* but also the SCC number M^c of the marking enabling *tf*, and the SCC number of its predecessor marking in the synchronisation graph (in a field *AncSCC*).

The communications are asynchronous. Thus, when receiving a message, the coordinator and worker processes are preempted and a handler function `MessageHandler`() is invoked. Algorithm 2 describes the actions performed by the coordinator when it receives a fused transition from the worker process s.

The parts handling other messages are not detailed. They concern the algorithm termination (and is similar to the termination in [KP04]), as well as the analysis messages introduced in section 4.

First, the coordinator creates the initial node $v_0 = M_0^c \in V_{SG}$. When a fused transition *tf* is received from a process s, the successors of a node $v = (v_1, v_2, \dots v_N) \in V_{SG}$ are computed as follows:

– we consider the set PM of the modules (processes) participating in the synchronisation of the fused transition *tf*;
– we construct the sets:
  • W_{pm} corresponding to the fused transition *tf* received earlier from other processes;
  • W_s corresponding to the fused transition received from process s;
  • W_{np} contains $\bullet$ for modules not participating in the synchronisation;
– for all possible combinations, the successor nodes in the synchronisation graph are calculated.

The successor states thus computed are sent to the worker processes for an additional round of internal state space generation.

4 Properties

A major issue is the analysis of concurrent systems properties. The reachability graph is the basic model on which most verifications are built. Behavioural properties, which depend on initial marking, and the reachability graph are often used to perform such analysis.

In this work we focus on basic behavioural properties: reachability, deadlocks, liveness, home state and their distributed analysis.

4.1 Reachability

The reachability problem for Petri nets consists in proving that a given marking M is in $[M_0\rangle$. This property is often used to check whether a faulty state M is reachable, e.g. an elevator moving while the door is open. It is convenient to look for erroneous states.

Algorithm 2: Message Handler()

```
1  begin
       ⋮
2      if $Message.Type =$ SynchTrans then
           /* message received contains an enabled fused transition */
3          $tf = Message.Content.tr$
4          $s = Message.Sender$
5          $PM = \{i \text{ s.t. } tf \in T_i\}$
           /* PM: modules participating in the synchronisation of $tf$ */
6          foreach $v = (v_1^c, v_2^c, \cdots, v_N^c) \in V_{SG}$ do
7              foreach $pm \in PM$ and $pm \neq s$ do
8                  $W_{pm} = \{w \in SGWaiting_{pm}$ s.t.
9                          $(w.AncSCC = v_{pm}^c) \wedge (w.tr = tf)\}$
                   /* $W_{pm}$: markings enabling $tf$, with $v_{pm}$ as ancestor */
10             foreach $np \notin PM$ and $np \neq s$ do
11                 $W_{np} = \{\bullet\}$
                   /* any marking of a non-participating module enables
                      synchronisation */
12             $W_s = \{Message.Content\}$
13             foreach $C = (c_1, c_2, \cdots, c_N)$ s.t. $\forall i \in PM, c_i \in W_i$ do
14                 $v' = (v'^c_1, v'^c_2, \cdots, v'^c_N)$ where:
15                         $\forall i \in PM, v'^c_i = c_i.SuccSCC$ and $\forall i \notin PM, v'^c_i = v^c_i$
16                 Add $(v')$
17                 AddArc $(v, (C, ft), v')$
18                 foreach $pm \in PM$ do
19                     Send $(Sender = Coordinator, Dest = pm, Type =$ NEW_NODE,
20                             $Content = v'_{pm})$
21         $SGWaiting_s = SGWaiting_s \cup Message.Content$
       ⋮
22 end
```

In some applications, one may be interested in the markings of a subset of places and not care about the others places in the net. This leads to a submarking reachability problem.

Reachability is known to be decidable. For modular systems, determining whether a given state M is reachable from the initial marking can be done in a parallel way: each *worker* process s searches through its local state space V_s. Then, if one of the worker processes does not find M_s in its local state space V_s, marking M is not reachable. Otherwise i.e. ($\forall s \in \{1..N\}\ M_s \in V_s$), all the workers send the ancestor SCCs of M_s to the coordinator. The *coordinator* stores the SCCs received from each worker s and checks whether there exists a combination of these in the set of nodes of the synchronisation graph V_{SG}.

4.2 Deadlocks

The algorithm used to find dead markings in a modular state space is based on the following proposition:

Proposition 1 (Deadlocks).

$$M \in [M_0\rangle \textit{ is a } \text{deadlock} \Leftrightarrow [\ \forall s \in S : (Ms)^c \in Term(SCC_s) \cap Trivial(SCC_s)) \\ \wedge(\forall (v_1, (M_1^c, tf), v_2) \in A_{SG} : M_1^c \neq M^c)]$$

Each worker process s searches in its local state space for the local dead markings which consist in trivial terminal SCCs. If there exists a worker process without such a node, the module assigned to this worker always allows a local behaviour. Hence, the system is deadlock-free. Otherwise, every worker process sends the SCCs corresponding to the locally dead markings to the coordinator process.

The coordinator process stores the SCCs received from the workers. It then checks each combination of such markings: if it labels an arc in the synchronisation graph, the corresponding fused transition is enabled and the marking is not a deadlock. Otherwise, if the marking is effectively reachable, then it is a deadlock.

4.3 Liveness

To verify whether a transition t is live or not, two cases are distinguished: if t is a fused transition ($t \in TF$), or t is an internal transition. The verification is based on the following proposition.

Proposition 2 (Liveness).

1. *A transition $tf \in TF$ is live $\Leftrightarrow$*
$$[\forall scc \in Term(SCC_{SG}) : tf \in Trans(scc)] \\ \wedge\ [\forall v \in V_{SG} : \forall M \in [[v\rangle : (\forall s \in S : M_s^c \in Term(SCC_s)) \\ \Rightarrow \exists (v, (M_1, tf'), v_2) \in A_{SG} : M_1 \in [[M\rangle].$$

2. A transition $t \in T_s$ is live $\Leftrightarrow$

$$\begin{aligned} &[\forall scc \in \mathit{Term}(SCC_{SG}) : \exists v \in scc : t \in \mathit{Trans}([v_s\rangle_s)] \\ \wedge\ &[\forall v \in V_{SG} : \forall M \in [[v\rangle : (M_s^c \in \mathit{Term}(SCC_s)) \\ &\quad \Rightarrow (t \in Trans(M_s^c) \vee \exists (v, (M_1, \mathit{tf}), v_2) \in A_{SG} : M_1 \in [[M\rangle))]. \end{aligned}$$

To check if a fused transition tf is live, the coordinator checks if there exists a terminal SCC in the synchronisation graph which does not contain transition tf, in which case transition tf is not live. Otherwise, for each node $v \in V_{SG}$ of the synchronisation graph, the coordinator sends v_s to the worker process s and waits to receive the terminal SCCs reachable from v_s in module s. Then, it checks whether the combinations of the terminal SCCs label fused transitions are in SG, considering only the modules participating in the synchronisation of this fused transition. If this is not the case, tf is not live.

When transition t is an internal transition of module s, the worker process s detects the terminal SCCs which do not enable t, that may invalidate liveness. For each node $v \in V_{SG}$ of the synchronisation graph, the coordinator sends v_s to the worker process s. The worker processes check whether these problematic SCCs are reachable from v_s, and send them to the coordinator. The coordinator checks if there exists a combination that does not label a fused transition, in which case transition t is not live.

4.4 Home States

The verification of whether a reachable state M_H is a home state or not is based on the following proposition.

Proposition 3 (Home state).
A state $M_H \in [M_0\rangle$ is a home state $\Leftrightarrow$

$$\begin{aligned} &[\forall scc \in \mathit{Term}(SCC_{SG}) : \exists v \in scc : M_H \in [[v\rangle] \\ \wedge\ &[\forall v \in V_{SG} : \forall M \in [[v\rangle : (\forall s \in S : M_s^c \in \mathit{Term}(SCC_s)) \\ &\quad \Rightarrow M_H \in [[M\rangle \vee \exists (v, (M_1, \mathit{tf}, M_2), v_2) \in A_{SG} : M_1 \in [[M\rangle]. \end{aligned}$$

To check such a property, we first check whether the state M_H is reachable from all nodes of the synchronisation graph. Then, for every node v of the synchronisation graph, the coordinator asks the worker processes for the terminal SCCs reachable from v by firing internal transitions only. The coordinator checks then that all combinations label an arc, or correspond to the SCC containing M_H. Otherwise M_H is not a home state.

5 Implementation and experiments

The previous algorithms were implemented within a prototype tool. The tests were carried on a cluster composed of 12 stations (Pentium IV with 512 Mbytes of memory), one of them is assigned to the coordinator process while the others run the worker processes.

In this section, we give the results obtained for the *dining philosophers problem* and *Automated Guided Vehicles (AGV) problem*, and draw conclusions.

5.1 The dining philosophers problem

The distributed generation based on partitioning the state space over a cluster of stations, allows for handling large size problems. In [BP07], the philosophers problem was handled with various sizes (see Table 1).

Nb Philo	Nb States	Nb Trans	Nb Cross. arcs	Nb Mess.	CPU Time (sec)
5	11	30	18	36	<0.01
10	123	680	242	494	<0.01
15	1,364	11,310	2,731	5477	0.06
20	15,127	167,240	30,236	60,493	2.52
25	167,761	2,318,400	315,718	631,587	32.58
30	1,728,813	28,686,031	3,572,821	7,156,116	7547.45

Table 1: Distributed generation based on partitioning the state space

But this approach is still limited. The total number of exchanged messages is very high, due to the amount of cross arcs.

In the distributed modular based approach, the original Petri net is split into a Modular Petri net with N modules and each module, which can contain m philosophers, is assigned to a *worker process*. Philosopher i of module l shares its forks with the philosophers $i-1$ and $i+1$ of the same module for $2 \leq i \leq m-1$. Philosophers 1 in module l and m in module $(l-1)\, mod\, N+1$ also share a fork.

The nature of the Petri net gives 4 SCCs in each module graph. The synchronisation graph size is 2^N nodes and $N.2^N$ arcs. For a problem with 100 philosophers, if we consider 10 modules with 10 philosophers each, we obtain 1,024 nodes and 10,240 arcs in the synchronisation graph, 233 nodes and 1,132 arcs in each local graph. The reachability graph size for the original problem is greater than 7×10^{20} nodes and 7×10^{22} arcs.

In Table 2, we give the modular state space sizes for the philosophers problem considering 10 philosophers per module each time. The average of CPU times of the coordinator is also given.

Nb Philo	Nodes N	Arcs	Mess Nb	CPU Time (sec)	CPU time (1 proc)
20	470	2162	26	0.04	0.08
40	948	4372	52	0.04	0.16
60	1462	6846	84	0.04	0.25
80	2120	10664	120	0.05	0.39
100	3354	21010	230	0.16	0.61

Table 2: Modular distributed generation

The CPU time of the workers is generally equal to 0.04 s (for 10 philosophers per module). Thus, in the first cases the global CPU time corresponds to the local state space generation, the synchronisation graph generation CPU time is less then 0.01 s. In the last cases the synchronisation graph generation spends more time since its size became larger.

Various tests were performed for reachability, liveness and home state properties, with the philosophers problem of 10 modules with 10 philosophers in each one, the experimental results are given in the table 3.

Properties	CPU Time (sec)	Mess Nb
Reachability1	< 0.01	20
Reachability2	0.03	20
Reachability3	0.02	20
Liveness ($Internal\ tr$)	0.17	4278
Liveness ($Fuzed\ tr$)	0.15	4116
Home state	0.21	4432

Table 3: Distributed modular verification of properties of the philosophers example

For reachability, we considered three cases: in the first, the marking is not reachable in some modules; in the second, the marking is reachable in the modules but not reachable as a global marking; and in the third case, the marking is reachable. The number of exchanged messages is the same and correspond to messages sent by the *coordinator* to the *worker* processes to transmit the marking to check (10 messages) and to the answers of the workers (10 messages). The liveness and home state verifications are performed with a relatively large number of exchanged messages: since all transitions are live, the coordinator must obtain the terminal SCCs reachable from each node of the synchronisation graph. When the coordinator moves from a node to its successor in the synchronisation graph, it transmits messages only to the workers corresponding to the changed components to have their terminal SCCs. This allows for minimising the number of messages transmitted by the coordinator.

5.2 Automated Guided Vehicles

The Automated Guided Vehicles (AGVs) problem has been solved by means of Modular State Spaces in [LP04]. The problem is that of a factory floor which consists of three workstations which operate on parts, two input and one output stations, and five AGVs which move parts from one station to another.

The 5 AGVs example is loosely coupled. Therefore, much interleaving is avoided when building the modular state space and this leads to very good results. This model has 30,965,760 states (see [LP04]). However, with modular analysis we obtain only 900 states with 2,687 arcs. The analysis based on modular distribution gives also good results as shown in table 4.

	CPU Time (sec)	Mess Nb
State Space Generation	0.02	82
Deadlocks search	0.03	11
Reachability1	< 0.01	22
Reachability2	0.02	22
Reachability3	0.02	22
Liveness ($Internal\ tr$)	0.09	3313
Liveness ($Fuzed\ tr$)	0.08	3062
Home state	0.15	3415

Table 4: Distributed verification of properties for the 5 AGVs example

6 Conclusion

In this paper, we proposed steps towards distributed modular analysis of Petri net models, based on the construction framework of [LP04]. Using these algorithms, it is possible to verify standard Petri nets properties, such as reachability, deadlocks, home states and liveness, in a distributed manner.

The main advantage of such an approach is the possibility to consider very large systems with several modules, sharing transitions, and assign them among a set of machines, thus limiting the drawbacks of the state space explosion problem.

Each machine (process) generates only the state space of the module it is assigned. The synchronisation graph provides the global knowledge of the behaviour of the system.

Moreover, it is also possible to check properties using the distributed modular state space directly, i.e. without unfolding to the ordinary state space. When designing algorithms there is often a trade-off between time and space complexity. For state space analysis it is attractive to have a rather fast way to decide properties, but the state space explosion problem makes it absolutely necessary to minimise memory usage.

Experiments were performed on a cluster of 12 stations. Both the AGVs, and the philosophers problems were considered, with different sizes. The results obtained for the generation of the modular state space were very interesting and allow for checking properties of very large systems. Future work will extend properties verification to temporal logic properties. Further experiments on larger case studies are also necessary for a better assessment of the benefits of this approach.

References

[AAC87] S. Aggarwal, R. Alonso, and C. Courcoubetis. Distributed reachability analysis for protocol verification environments. In P. Varaiya and H. Kurzhanski, editors, *Discrete Event Systems: Models and Application*, volume 103 of *LNCIS*, pages 40–56. Springer-Verlag, 1987.

[AHI98] K. Ajami, S. Haddad, and J.-M. Ilié. Exploiting symmetry in linear temporal model checking: One step beyond. In *Proc. 4th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 52–67, 1998.

[BBS01] J. Barnat, L. Brim, and J. Strîbrnà. Distributed LTL model checking in SPIN. In *Proc. 8th International SPIN workshop on Model Checking Software (SPIN'2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 200–216. Springer-Verlag, 2001.

[BO03] S. Blom and S. Orzan. Distributed state space minimization. *Electronic Notes in Theoretical Computer Science*, 80:1–15, 2003.

[BP07] M.C. Boukala and L. Petrucci. Towards distributed verification of Petri nets properties. In British Computer Society eWIC, editor, *In Proc. 1st International Workshop on Verification and Evaluation of Computer and Communication Systems (VECOS'07), Algiers, Algeria*, pages 15–26, 2007.

[CFJ93] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Proc. 5th Int. Conf. Computer Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 450–462. Springer-Verlag, 1993.

[GMS01] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. In *Proc. 8th International SPIN workshop on Model Checking Software (SPIN'2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 217–234. Springer-Verlag, 2001.

[KP04] L. Kristensen and L. Petrucci. An approach to distributed state exploration for coloured Petri nets. In *Proc. 25th Int. Application and Theory of Petri Nets (ICATPN'2004)*, volume 3099 of *Lecture Notes in Computer Science*, pages 474–483. Springer-Verlag, 2004.

[LP04] C. Lakos and L. Petrucci. Modular analysis of systems composed of semiautonomous subsystems. In *In Proc. 4th Int. Conf. on Application of Concurrency to System Design (ACSD'2004), Hamilton, Canada.*, pages 185–194. IEEE Computer Society Press, 2004.

[LS99] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. 5th and 6th International SPIN workshops on Model Checking Software (SPIN'1999)*, volume 1680 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1999.

[NG97] R. Nalumasu and G. Gopalakrishnan. A new partial order reduction algorithm for concurrent systems. In *Proc. Int. Conf. Hardware Description Languages and their Applications (CHDL'97)*. Chapman & Hall, 1997.

[Sch04] K. Schmidt. Automated generation of a progress measure for the sweep-line method. In *Proc. 10th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 192–204. Springer-Verlag, 2004.

[Val92] A. Valmari. A stubborn attack on state explosion. *Formal Methods in Systems Design*, 1(4):297–322, 1992.