Local Verification Using a Distributed State Space

Chiheb Ameur Abid*, Belhassen Zouari
LIP2 Laboratory
University of Tunis - El Manar 2092
Tunis, Tunisia
chiheb.abid@gmail.com; belhassen.zouari@fst.rnu.tn

Abstract. This paper deals with the modular analysis of distributed concurrent systems modelled by Petri nets. The main analysis techniques of such systems suffer from the well-known problem of the combinatory explosion of state space. In order to cope with this problem, we use a modular representation of the state space instead of the ordinary one. The modular representation, namely modular state space, is much smaller than the ordinary state space. We propose to distribute the modular state space on every machine associated with one module. We enhance the modularity of the verification of some local properties of any module by limiting it to the exploration of local and some global information. Once the construction of the distributed state space is performed, there is no communication between modules during the verification.

Keywords: Distributed systems, modular verification, Petri nets, state space explosion

1. Introduction

A factory automation system is usually viewed as a network of machines which perform local operations, and which interact periodically in some way. A malfunctioning machine with operational neighbours can disturb the production, and a starvation phenomenon or blocking may occur. Besides, the integration of new subsystems or updating some already existing subsystems must be done in consistent with the reality way in order to produce the planned results for the overall system. Indeed, each subsystem may be well-modelled and extensively tested, however putting the subsystems in interaction can produce unexpected behaviours [15]. Since constraints inherent in the synchronisation can affect behaviours of modules.

* Address for correspondence: LIP2 Laboratory - University of Tunis - El Manar 2092, Tunis, Tunisia
**Petri nets**, as a formal tool, have been used for many years to model and analyse such systems. They have proved powerful on modelling, analysis, control, optimization, simulation, and implementation of various engineering systems. Given the flexible nature of engineering systems, it is necessary to guarantee that they behave correctly. The analysis of such systems can present a real challenge. Even with simple configurations, the generated state space of a system grows exponentially with number of states. This phenomenon is the well-known *state explosion problem*, can be overcome by a modular representation of the state space. Several methods have been developed which exploit such representation for the verification of concurrent systems. Among them, we cite algebraic process [19, 20] and modular analysis of Petri nets [6, 13, 16] methods. Algebraic process methods are mainly based on compositional approach: the local state spaces are progressively reduced and composed until the obtaining of only one reduced graph. Finally, the reduced graph is used to check preserved properties by the execution of classic algorithms based on its exploration. The modular analysis methods permit a modular verification of interesting properties on the basis of a modular representation of the state space. This representation, called modular state space, consists of two different structures of graphs, namely local graphs and synchronisation graph. A local graph of a module represents only local reachable markings to the module and the occurrences of its local transitions. The synchronisation graph is used to store the occurrences of synchronised actions between modules. In addition to the compact representation of the state space, the modular analysis method avoids the enumeration of all global markings during the verification of some interesting properties. Indeed, modular verification allows to downsize the analysis of a system to the analysis of its individual modules [11]. However, using modular verification is difficult. This is because verifying a property of module in isolation often requires to introduce its environment, so that the module is not completely free in its interaction with environment. This problem is called the *environment problem* [17].

Recently, there has also been increased interest in distributed verification approaches [1, 4, 5, 7, 8, 9, 12]. A distributed approach does not alleviate the explosion problem, however it offers more available memory for storage of the state space and has the potential to speed-up the process of verification in time. One can easily notice that the philosophy of a distributed approach fits well with a modular approach. In this context, *modular Petri nets* [6, 10, 13] define an important framework for investigating modular/distributed systems [2, 18, 21]. These models introduce structure to Petri nets by letting modules be specified separately, and include the communication mechanisms between modules.

In this work, we propose a distributed verification approach for modular systems modelled by Petri nets. Our aim is to allow the analysis of some properties of a module in isolation by exploring its local information and some global information. This approach is based on our previous work of [1]. We consider a modular system modelled by a modular Petri net. We build its modular state space in a distributed manner. For this, we associate a process (machine) with every module, and an extra process to build the synchronisation graph. Each process of those associated with modules has to maintain the local state space of its associated module, and a reduced copy of the synchronisation graph. Further, the local state spaces are enriched with information indicating whether a module reaches a state from which none of its synchronised actions can be executed, or not. Such information is determined during the building of the modular state space, and it allows to check some properties of a module by exploring its local state space and its reduced copy of the synchronisation graph. Therefore, the verification of a property is performed without more communication with other modules, i.e. *local verification*.

The paper is scheduled as follows: in section 2, we recall the definitions of modular Petri nets and modular state spaces; in section 3, we introduce the Sync-closure property, and we describe the technique
allowing to compute it; in section 4, we present how to reduce a synchronisation graph according to a module; in section 5, we illustrate how to check a local property of one module; in section 5 we propose a distributed approach for the construction of the modular state space; section 6 summarizes the main conclusions and perspectives of this work.

2. Preliminaries

In many domains, modularity is a key principle to deal with the design of complex systems. Indeed, modularity provides several advantages. It allows to specify modules separately and reusability. It has been argued that the computer industry has dramatically increased its rate of innovation by adopting modular design [3]. In this context, modular Petri nets are efficient models used for the specification of modular systems. This section contains the basic definitions and notations for modular Petri nets, their marking, enabling and occurrence rules. Most of these notions are taken from [6, 13].

Definition 2.1. A Petri net is a tuple \( PN = (P, T, W) \), where \( P \) is a finite set of places, \( T \) is a finite set of transitions such that \( T \cap P = \emptyset \), \( W \) is the arc weight function mapping from \( (P \times T) \cup (T \times P) \) into \( \mathbb{N} \).

Definition 2.2.

- A marking \( M \) of a Petri net \( PN = (P, T, W) \) is a mapping \( M : P \rightarrow \mathbb{N} \), \( M(p) \) denotes the number of tokens contained in place \( p \).
- The tuple \( (P, T, W, M_0) \) is called a marked Petri net and \( M_0 \) is the initial marking.
- A transition \( t \) is enabled in a marking \( M \), denoted by \( M[t] \), iff \( \forall p \in P : W(p, t) \leq M(p) \).
- If \( M[t] \), the transition \( t \) may occur, changing the marking \( M \) to another marking \( M' \), defined by:
  \[
  \forall p \in P : M'(p) = (M(p) - W(p, t)) + W(t, p).
  \]
- The set of reachable markings from a marking \( M \), is: \( [M] = \{ M' | \exists \sigma \in T^* : M[\sigma]M' \} \).

Modular Petri nets introduce structure to Petri nets by letting modules be specified separately. The modules communicate either by using shared transitions or places. In this paper, we consider only modules which communicate through shared transitions. Christensen and Petrucci [6] have shown that modular nets with shared places can be transformed to nets using only shared transitions.

Definition 2.3. A modular Petri net is a pair \( MN = (S, \mathcal{F}) \), satisfying:

1. \( S \) is a finite set of modules such that:
   (a) Each module, \( s \in S \), is a Petri net \( s = (P_s, T_s = T_{\text{sync}, s} \cup T_{\text{l}, s}, W_s, M_{0_s}) \), \( T_{\text{sync}, s} \) denotes the synchronised (fused) transitions set of module \( s \).
      \( T_{\text{l}, s} \) denotes the local (internal) transitions set of module \( s \).
   (b) The sets of transitions (resp. places) corresponding to different modules are pair-wise disjoint: \( \forall s_1, s_2 \in S, s_1 \neq s_2 \Rightarrow T_{s_1} \cap T_{s_2} = \emptyset \) (resp. \( P_{s_1} \cap P_{s_2} = \emptyset \)),
(c) $P = \bigcup_{s \in S} P_s$ and $T = \bigcup_{s \in S} T_s$ the set of all places and all transitions of all modules.

2. $\mathcal{F} \subseteq 2^T \setminus \{\emptyset\}$ is a finite set of non-empty transition fusion sets.

In the following, $T_{\text{sync}}$ denotes the set of all synchronised transitions, i.e. $T_{\text{sync}} = \bigcup_{f \in \mathcal{F}} f$. $T_l$ denotes the set of all local transitions (non synchronised transitions), i.e. $T_l = T \setminus T_{\text{sync}}$. A transition fusion set is a set of synchronised transitions belonging to different modules which firing represents the execution of a synchronised action.

In order to express the enabling and occurrence rules of a modular Petri net, we present the notion transition groups. It allows to take into account the two kind of transitions, namely the local transitions and the synchronised ones.

**Definition 2.4.** A transition group $tg \subseteq T$ consists of either a single non fused-transition $t \in T_l$ or all members of a transition fusion set $f \in \mathcal{F}$. The set of transition groups is denoted by $\mathcal{TG}$.

Next, we extend the arc weight function $W$ to transition groups:

$$\forall p \in P, \forall tg \in \mathcal{TG} : \begin{align*} W(p, tg) &= \sum_{t \in tg} W(p, t), \\ W(tg, p) &= \sum_{t \in tg} W(t, p). \end{align*}$$

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 $M_s$, i.e. $M_s$ is the projection of $M$ on $s$. $M_s$ is a local marking of module $s$. The enabling and occurrence rules of a modular Petri net can now be extended to transition groups.

**Definition 2.5.** Let $MN$ be a modular Petri net modelling a set of modules. A transition group $tg$ is enabled in a marking $M$, denoted $M[tg]$, iff $\forall p \in P : W(p, tg) \leq M(p)$. When a transition group $tg$ is enabled in a marking $M$ it may occur, changing the marking $M$ to another marking $M'$, defined by: $\forall p \in P, M'(p) = (M(p) - W(p, tg)) + W(tg, p)$.

**Example 2.1.** Figure 1 gives a modular Petri net. It consists of three modules $A$, $B$ and $C$. Modules $B$ and $C$ both are synchronised on transition $\text{Sync}1$, while modules $A$ and $C$ both are synchronised transition $\text{Sync}2$. These transitions are assumed to form two transition fusion sets.
Now, we give the definition of modular state space. For this, we denote the set of states reachable from \( M \) by occurrences of local transitions only, in all the individual modules, by \([M]\). The notation with a subscript \( s \) means the restriction to module \( s \), e.g. \([M]_s\) is the set of all nodes reachable from \( M \) by occurrences of transitions in module \( s \). Let \( S \) be the set of modules, a global marking \( M \) is the product of local markings of individual modules, i.e. \( M = \prod_{s \in S} M_s \). Indeed, \( \forall s \in S, M_s \) is the restriction of \( M \) to module \( s \). When considering a modular state space, as well as checking properties of the system, we will use Strongly Connected Components (SCC). In a local graph, the SCC of a local marking \( M_s \) is the maximal set of markings, denoted by \( M^c_s \), such that \( \forall M_1, M_2 \in M^c_s \), there is a path from \( M_1 \) to \( M_2 \). A product of SCCs of individual modules is denoted by \( M^\varphi = \prod_{s \in S} M^c_s \).

The main idea of the modular state space structure is to deal with two kinds of graphs, namely the local graphs and the synchronisation graph. Typically, a local graph describes the evolution of a module in terms of local transitions, whereas the synchronisation graph illustrates the interactions between modules based on synchronised transition firings.

**Definition 2.6.** Let \( M_0 \) be the initial marking of a modular Petri net \( MN \) modelling a set \( S \) of modules, and \( s = (P_s, T_s, W_s, M_{0_s}) \) be a Petri net of a module in \( S \). The local graph of \( s \) is a tuple \((Q_s, A_s)\), such that:

- \( Q_s = \{M_s | M \in [M_0]\} \)
- \( A_s = \{(M_s, t, M'_s) | M \in [M_0] \land t \in T_{1,s} \land M_s[t]M'_s\} \)

The synchronisation graph \( SG \) of \( MN \) is a couple \((Q_{SG}, A_{SG})\), where \( Q_{SG} \) is the set of its nodes and \( A_{SG} \) is the set of its arcs:

- \( Q_{SG} = \{M_0^\varphi \} \cup \{M'^\varphi | \exists M' \in [M_0], \exists f \in F: M'[f]M\} \)
- \( A_{SG} = \{(M'^\varphi, (M'^\varphi, f), N'^\varphi) | M'^\varphi, N'^\varphi \in Q_{SG} \land f \in F \land M'^\varphi \in [M] : M'[f]N\} \)

Let \((-,-,-,-)\) be an arc in the synchronisation graph. We call \( q' \) the start product of the considered arc. The symbol “−” indicates any possible value of the arc component.

**Example 2.2.** Figure 2 shows the modular state space of the modular Petri net described in Fig. 1. There are three local graphs and one synchronisation graph. The nodes of each local graph are the local markings, and the arcs linking them correspond to the occurrences of local transitions realised between linked nodes. We do not make distinction between SCCs and local markings, since in this example each SCC contains only one local marking. Further, the number of arcs and nodes of the synchronisation graph is optimised by representing equivalent occurrences of transition fusion sets. This optimisation is performed by replacing SCCs of modules which are not synchronised on a synchronised transition by the symbol \( \varnothing \) in arcs representing the transition occurrences. Indeed, the firing of a synchronised transition \( tsync \) does not have any effect on modules which are not synchronised on the transition \( tsync \).

### 3. Sync-closure property

In order to be able to check local properties of one module by exploring its local graph and the synchronisation graph without the need to explore the local information of other modules, we have introduced the
property of Sync-closure [1]. As an important result, this new property improves the modularity of the verification of local properties of modules. This improvement consists in restricting the verification of a local property of a module to the exploration of its local information (local graph) and global information (synchronisation graph).

In this work, our aim is to further improve the modularity of the verification by limiting the global information to explore. More precisely, our purpose is to consider only global information relevant to the module concerned by the local property to check. Instead of exploring the complete synchronisation graph, we propose to use a reduced copy of the synchronisation graph associated with the concerned module. Intuitively, a reduced copy of the synchronisation graph associated with a module represents the interactions of the module with its environment. Therefore, we propose to extend the Sync-closure property in order to achieve this purpose.

3.1. Formal definitions of a Sync-closed SCC

Intuitively, the sync-closure property when it holds for an SCC of a module in a modular system, it indicates that the module of the system eventually reaches a state from which it evolves only by executing local transitions. Without the sync-closure property, the exploration of the local graph of a module and the synchronisation graph does not suffice to know whether the module reaches a state from which it can not evolve. For instance, by exploring the local graph of module A and the synchronisation graph, illustrated by Fig. 2, we can not conclude that module A can stop evolving from marking A2. This is because the synchronisation graph shows that the synchronised transition sync2 is always fired in marking A2 to reach marking A1, and then fire transition ta to reach again A2. Indeed, these two graphs do not show that the whole system stops evolving when it reaches the global state A2B3C2 which is a deadlock global marking. Therefore, module A stops evolving when it reaches A2.

First, we introduce the totally sync-closure property for terminal SCCs. A terminal SCC $c_s$ of a module $s$ is called a totally Sync-closed when it contains a local marking $M_s$ such that from the global marking $M^0$ (built from $M_s$) only local transitions can be fired.

**Definition 3.1.** Let $M_0$ be the initial marking of a modular Petri net $MN = (S, F)$. Let $c_s$ be a terminal SCC of a module $s$. $c_s$ is a totally Sync-closed SCC if and only if $\exists M \in [M_0]$ such that $M_0^0 = c_s$ and $\forall f \in F, \forall M' \in [M] : \neg M'[f]$.

**Remark 3.1.** A totally Sync-closed SCC could participate in two different global markings, one of them allows to fire a synchronised transition and the other not. This has no influence in the verification process, since the provided information indicates that the whole system will eventually reach a state from which no synchronised transition will be fired, i.e. every module will eventually evolve only locally.

**Example 3.1.** In the modular state space of Fig. 2, there are three totally Sync-closed SCCs A2, B2 and C1. Indeed, the global marking $A2B2C1$ does not allow any synchronised transition to fire. This means, for instance, when the module A reaches the local marking A2, it will only evolve locally (by occurrences of local transitions only).

In our approach, the local verification of a module property is based on the exploration of the local graph associated with the module and a reduced copy of the synchronisation graph. The reduction is
performed by hiding the firings of synchronised transitions on which the module is not synchronised. For this purpose, we extend the definition of the sync-closure property in order to capture when the system can reach a state from which only synchronised transitions of other modules can be fired. We introduce the notion of partially Sync-closed SCCs. Intuitively, a partially Sync-closed SCC of a module $s$ indicates that the module $s$ can not evolve by executing synchronised transitions, while the other modules can. A terminal SCC $c_s$ of a module $s$ is called a partially Sync-closed SCC if and only if, it is totally Sync-closed or, it contains a local marking $M_s$ from which we can build a global marking $M$ such that from $M$ no synchronised transition of module $s$ is fireable.

**Definition 3.2.** Let $M_0$ be the initial marking of a modular Petri net $MN = (S, \mathcal{F})$. Let $T_{sync,s}$ be the set of synchronised transitions of a module $s$. Let $c_s$ be a terminal SCC of $s$. $c_s$ is a partially Sync-closed SCC iff $\exists M \in [M_0]$ such that $M_s^c = c_s \land \forall M' \in [M] : M(\{tsync\}) \Rightarrow tsync \cap T_{sync,s} = \emptyset$.

We can easily remark that partially Sync-closure property corresponds to a weaker condition than the one of totally Sync-closure property. Indeed, a totally Sync-closed SCC $c_s$ implies that $c_s$ is partially Sync-closed.

**Property 1.** Let $c_s$ be a SCC of a module $s$. $c_s$ is a totally Sync-closed SCC $\Rightarrow$ $c_s$ is a partially Sync-closed SCC.

**Remark 3.2.** In case of a module $s$ has a partially Sync-closed SCC $c_s$ which is not totally Sync-closed, it means that the module will evolve only by executing its local transitions from any marking of $c_s$. However, the other modules are able to fire any synchronised transition which is not synchronised with module $s$.

**Example 3.2.** In the modular state space of Fig. 2, there are four Sync-closed SCCs. They are represented by bold ellipses. As an example, $C2$ is a partially Sync-closed SCC which is not totally Sync-closed. This indicates that module $C$ can only evolve by the occurrences of local transitions when it reaches the state $C2$, while the other modules may be able to execute synchronised transitions.
3.2. Computing Sync-closed SCCs of a module

The determination of Sync-closed SCCs of one module is performed just after the building of the modular state space through two steps. First, we determine the totally sync-closed SCCs, then we compute partially Sync-closed SCCs from remaining ones.

The determination of totally Sync-closed SCCs can be performed by checking whether all possible products of terminal SCCs, locally reachable from the same node of synchronisation graph, enable a transition fusion set, or not. This requires the computation of the Cartesian product of terminal SCCs of individual modules. The complexity of Cartesian product is at worst $O(n^m)$ where $n$ denotes the number of SCCs in a local graph, and $m$ denotes the number of modules (i.e. $m = |S|$). Here, we propose an efficient technique for computing Sync-closed SCCs without computing Cartesian products allowing to reduce the complexity to at worst $O(n)$. Mainly, the cost of our technique comes from the exploration of local graphs to determine terminal SCCs.

Further, we emphasize that the determination of the Sync-closed SCCs is performed only once before checking properties, while algorithms proposed in [6, 13] require that the computation of Cartesian products is carried out at each operation of verifying a property.

Let $q$ be a node of the synchronisation graph. Let $c_s$ be a terminal SCC of a module $s$ locally reachable from $q$. If we can build from $c_s$ a product of SCCs locally reachable from $q$ such that the product is not labelling any outgoing arc from the node $q$ in the synchronisation graph, then $c_s$ is totally Sync-closed. It means that the number of products of terminal SCCs locally reachable from $q$ is strictly inferior than the number of those enabling a transition fusion set. Therefore, we exploit this inequality to check whether an SCC of a module is totally Sync-closed or not. It is performed through the following four steps for every node $q$ of the synchronisation graph:

1. We consider a terminal SCC $c_s$ of module $s$ locally reachable from $q$.
2. Let $\text{Term}_{q;i}$ be the set of terminal SCCs locally reachable from $q$ in module $i \in S$. We compute the product $\prod_{i \in S \setminus \{s\}} |\text{Term}_{q;i}|$. This product is the number of the possible products built from $c_s$ and the terminal SCCs of other modules locally reachable from $q$.
3. By considering the outgoing arcs from the node $q$, we determine the start products built from $c_s$.
These start products represent the products of terminal SCCs enabling a transition fusion set.
4. If the two computed numbers are distinct, we qualify the SCC $c_s$ as a totally Sync-closed SCC. Otherwise, we do nothing.

Let us consider a synchronisation graph $SG = (Q_{SG}, A_{SG})$ of a modular Petri net $MN$ modelling a set $S$ of modules. In the following, we propose to introduce a partial order denoted by $\leq$ relation defined on a set of start products. We recall that a start product represents a set of SCC products due to the presence of symbol $\omega$. Let $q$ and $q'$ be two start products, $q \leq q' \iff \forall s \in S, q_s = q'_s \vee q'_s = \omega$.

Let $E$ be a set of start products. Each element of $E$ represents a subset of SCC products enabling a transition fusion set. An element of $E$ can be represented by another, i.e. the subset is included in another. For this, we determine from $E$ elements which are not represented by others. Formally such set is obtained as follow:

$$\max(E) = \{ q \in E / \exists q' \in E : q \leq q' \}$$
By considering the set $\text{max}(E)$, we can determine the number of SCC products which contain a marking enabling a transition fusion set. For every element of $E$,

Let $(q, (q', -), -)$ be an arc of the synchronisation graph $SG$.

For every $i \in S$, the number of terminal SCCs represented by $q'_i$ is

$$R(q'_i) = \begin{cases} 
|\text{Term}_{q,i}| & \text{if } q'_i = \omega, \\
1 & \text{if } q'_i \text{ is a terminal SCC}, \\
0 & \text{otherwise.}
\end{cases}$$

Therefore, the number of products of terminal SCCs locally reachable from $q$ represented by the start product $q'$ is:

$$R(q') = \prod_{s \in S} R(q'_s)$$

The following proposition recapitulates the necessary condition allowing to decide whether an SCC local to a module is totally Sync-closed.

**Proposition 3.1.** Let $s$ be a module in $S$, $q$ be a node in $Q_{SG}$ and $c_s$ be a terminal SCC of module $s$ locally reachable from $q$.

Let $E = \{M^q/(q, (M^q', -), -) \in A_{SG} \land (M^q_s = c_s \lor M^q_s = \omega)\}$. If:

$$\sum_{M^q \in \text{max}(E)} R(q(M^q)) < \prod_{i \in S \setminus \{s\}} |\text{Term}_{q,i}|$$

then $c_s$ is a totally Sync-closed SCC, where $\text{Term}_{q,i}$ is the set of terminal SCCs of module $i$ locally reachable from $q$.

Now, we consider the computation of partially Sync-closed SCCs which are not totally Sync-closed. From the definition of a partially Sync-closed SCC, we derive the following proposition allowing the computation of a such SCC.

**Proposition 3.2.** Let $M_0$ be the initial marking of a modular Petri net $MN = (S, F)$. Let $T_s$ be the set of transitions of a module $s \in S$.

Let $c_s$ be a terminal SCC of module $s$ such that $c_s$ is not totally Sync-closed.

$c_s$ is a **partially Sync-closed** SCC iff $\exists \text{sec} \in \text{TermSCC}_{SG} : (\forall f \in \text{Trans}(\text{sec}), f \cap T_s = \emptyset) \land (\exists q \in \text{sec} : c_s \text{ is locally reachable from } q)$ where $\text{TermSCC}_{SG}$ denotes the set of terminal SCCs of the synchronisation graph and $\text{Trans}(\text{sec})$ is the set of transitions labelling arcs of $\text{sec}$.

**Proof:**

Let $c_s$ be a terminal SCC of a module $s$ such that $c_s$ is partially Sync-closed.

Then, from the definition of a partially Sync-closed SCC, the markings of $c_s$ are locally reachable from a node $q1$ of the synchronisation graph such that only synchronised transitions, on which $s$ is not synchronised, are fireable from $q1$. Therefore, in the synchronisation graph, all nodes are reachable from $q1$ by the firing of synchronised transitions which are not synchronised on $s$. Further, $c_s$ is locally reachable from any of these nodes, since no synchronised transition of module $s$ is fired. Thus, it exists a node $q$ belonging to a terminal SCC of the synchronisation graph such that all of its arcs correspond to the firing
Procedure \textit{ComputeSyncClosedSCC}(NodeSG \, q, \, Module \, s, \, \forall i \in S : \text{Term}_{q,i} \text{ is the set of terminal SCCs of module } i \text{ locally reachable from } q): \text{ a Set of SCCs}

\begin{align*}
&\text{SyncClosedSCC}_s \leftarrow \emptyset \\
&\text{/*Determining totally Sync-closed SCCs*/} \\
&T \leftarrow \prod_{i \in S \setminus \{s\}} |\text{Term}_{q,i}| \\
&\text{forall the } c_s \in \text{Term}_{q,s} \text{ do} \\
&E \leftarrow \{M^e / (q, (M^e, -), -) \in A_{SG} \land (M^e_s = c_s \lor M^e_s = \omega)\} \\
&\text{if } \sum_{M^e \in \max(E)} R_q(M^e) < T \text{ then} \\
&\text{SyncClosedSCC}_s \leftarrow \text{SyncClosedSCC}_s \cup \{c_s\} \\
&\text{else} \\
&\text{if } \exists \text{scc } \in \text{SCC}_{SG} \text{ s.t. scc is terminal and } \text{Trans(scc)} \cap T_s = \emptyset \text{ then} \\
&\text{if } c_s \text{ is locally reachable from scc} \text{ then} \\
&\text{SyncClosedSCC}_s \leftarrow \text{SyncClosedSCC}_s \cup \{c_s\} \\
&\text{return } \text{SyncClosedSCC}_s
\end{align*}

Figure 3. Determining a set of Sync-closed SCCs of a module

of synchronised transition not belonging to module \(s\).

The proof of \(\Leftarrow\) is similar to \(\Rightarrow\). \(\square\)

In brief, every non totally Sync-closed terminal SCC \(c_s\) verifying the two following requirements is called a partially Sync-closed SCC.

- \(c_s\) is locally reachable from a node \(q\) of synchronisation graph such that \(q\) belongs to a terminal SCC \(\text{scc}_{SG}\) of the synchronisation graph,
- the arcs linking the nodes of \(\text{scc}_{SG}\) represent the occurrences of synchronised transitions not belonging to module \(s\).

Given a node \(q\) of the synchronisation graph and a module \(s\), the procedure \textit{ComputeSyncClosedSCC} of Fig. 3 computes the set of Sync-closed SCCs of module \(s\) locally reachable from \(q\). We use \(\text{Term}_{q,i}\) to denote the set of terminal SCCs locally reachable from \(q\) in a module \(i\), where \(q\) is a node in the synchronisation graph. The function \(\text{Trans}\) maps a set of nodes to the set of transitions occurring in the labels of arcs linking two nodes of the given set. First, for every terminal \(c_s\) in the module \(s\), the procedure \textit{ComputeSyncClosedSCC} checks whether \(c_s\) is totally Sync-closed, or not. If it is not, it checks whether it is a partially Sync-closed SCC by verifying the condition of Proposition 3.2.

\textbf{Example 3.3.} Using the procedure of Fig. 3, our aim is to determine the Sync-closed SCCs of module \(C\). First, we consider the the node \(A1B1C1\). From this node, there is one terminal SCC \(C1\). The number of products built from terminal SCCs locally reachable from \(A1B1C1\) is \(\mathcal{T} = 1 \ast 2 \ast 1 = 2\). Further, there is one outgoing arc from \(A1B1C1\) in the synchronisation graph. This arc is labelled with \((\omega B3C1, Sync1)\). The number of terminal SCC products represented by \((\omega B3C1, Sync1)\) is
\[
\mathcal{R}(A1B1C1, \omega B3C1) = 1 \ast 1 \ast 1 = 1
\]
Then, we have \( R(A1B1C1, \omega B3C1) < T \). Consequently, \( C1 \) is a totally Sync-closed SCC. Thus, \( C1 \) is a Sync-closed SCC.

By applying the procedure for the node \( A1B4C2 \) of the synchronisation graph, we can conclude easily that \( C2 \) is not a totally Sync-closed SCC. Then, we have to check the second part of the condition to decide whether \( C2 \) is a partially Sync-closed SCC, or not. There exists one terminal SCC in the synchronisation graph, namely \( A1B4C2 \), from which the marking \( C2 \) is locally reachable. Therefore, \( C2 \) is a partially Sync-closed. Thus, \( C2 \) is a Sync-closed SCC.

4. Reduction of the synchronisation graph according to one module

Considering one module of a modular Petri net, the firing of any synchronised transition on which the module is not synchronised can be seen by the module as a local transition. Indeed, the firing of a synchronised transition affects only the local markings of modules synchronised on the transition. Based on this observation, we propose to reduce the synchronisation graph according to every module. Then, the verification of a local property of a module will be performed by exploring its local graph and the reduced copy of the synchronisation graph according to this module. It is worth noting that this operation of reduction must be performed after the computation of Sync-closed SCCs.

Let us consider a module \( s \) where \( T_s \) is the set of its transitions. Let \( f \) be a transition fusion set such that \( f \cap T_s = \emptyset \). Then, the firing of \( f \) can be viewed for the module \( s \) as the firing of a local transition of another module. This implies that we can abduct the representation of the firings of \( f \) in the synchronisation graph when considering module \( s \). Thus, arcs and nodes induced by the firings of \( f \) can be removed from the synchronisation graph.

In order to reduce the synchronisation graph according to a module \( s \), we start by renaming transitions on which module \( s \) is not synchronised by \( \tau \). Then, we proceed according to the following rules:

- Every arc of the synchronisation graph labelled with \( \tau \) linking a node \( q \) to a node \( q' \) is removed. Further, the successors nodes of \( q' \) must be assigned as successors of \( q \). This is performed by adding arcs linking node \( q \) to the successors of \( q' \).

- After removing all arcs labelled with \( \tau \), we remove unreachable nodes and their arcs in synchronisation graph.

Algorithm of Fig. 4 describes the reduction of a copy of the synchronisation graph according to one module \( s \). It starts by renaming the synchronised transitions on which module \( s \) is not synchronised by \( \tau \). Then, each arc labelled with \( \tau \) is removed. The successors of the destination node associated with the removed arc are assigned as successors for the source node of the same removed arc. Finally, unreachable parts of the graph are removed.

Example 4.1. We consider the synchronisation graph described in Fig. 2 in order show the operation of reduction. We apply the algorithm of Fig. 4 for the three modules forming the system. We obtain the three reduced synchronisation graphs as it is illustrated by Fig. 5.
**Procedure** ReduceSG($Q_{SG}$ a set of nodes, $A_{SG}$ a set of arcs, Module $s$

forall the $(q, (-, f), q') \in A_{SG}$ do

if $f \cap T_s = \emptyset$ then

$A_{SG} \leftarrow A_{SG} \setminus \{(q, (-, f), q')\}$

$A_{SG} \leftarrow A_{SG} \cup \{(q, \tau, q')\}$

while there exists an arc in $SG$ labelled with $\tau$ do

Let $(q, \tau, q')$ be an arc of $SG$

for every $(q', (q_1, f), q'')$ output arc of $q'$ do

$A_{SG} \leftarrow A_{SG} \setminus \{(q', (q_1, f), q'')\}$

$A_{SG} \leftarrow A_{SG} \cup \{(q, (q_1, f), q'')\}$

$A_{SG} = A_{SG} \setminus \{(q, \tau, q')\}$

Remove from $Q_{SG}$ all unreachable nodes from $q_0$

Figure 4. Reducing a synchronisation graph according to one module

Figure 5. Reduced synchronisation graphs

5. **Local properties verification**

Our proposed algorithms for the verification of local properties of a module are based on those presented in [6, 13]. By taking advantage of the Sync-closure property, the modularity of the verification is improved, since the verification of a local property requires only the exploration of the local graph of the module and its reduced copy of the synchronisation graph. Indeed, initial algorithms proposed in [6, 13] require the exploration of all local graphs, even though if that the considered property concerns only one module. Indeed, these algorithms repeat the same computations in order to determine some products of SCCs containing Sync-closed SCCs. Such a computation is repeated for each operation of verification of a property. It consists first, in determining the set of terminal SCC products. Then, products which do not label arcs of the synchronisation graph are removed. The use of the Sync-closed SCC concept improves these algorithms on two levels. First, the computation of Sync-closed SCCs is not repeated at each operation of verification. Secondly, it enhances the modularity as it avoids exploring local graphs of modules which are not concerned by the property to check.
In this work, we focus on the verification of three local properties of concurrent systems, namely liveness, deadlocks and home space. For this, we consider a modular Petri net $MN = (S, F)$ which consists of a set $S$ of Petri nets such that $\forall s \in S$, $s = (P_s, T_s, W_s, M_{0s})$. The global initial marking of $MN$ is $M_0$. Let $SG = (Q_{SG}; A_{SG})$ be the reduced synchronisation graph of $MN$. We use the function $Trans$ to map a set of nodes to the set of transitions occurring in the labels of arcs linking two nodes of the set. We use $Term$ to map a set of SCCs to set of its terminal SCCs. $SCC_{SG}$ denotes the set of SCCs of the synchronisation graph, and $SCC_s$ denotes the SCCs of the local graph associated with a module $s \in S$.

5.1. Liveness

Here, we are interested in checking the liveness of a set of transitions belonging to the same module. A transition is said to be live if no matter what firing sequence has occurred so far, the transition can eventually fire again. Now, we give the definition of the liveness for a set of transitions belonging to the same module.

Definition 5.1. Let $X$ be a set of transitions belonging to the same module $s \in S$. $X$ is live iff $\forall M \in [M_0] : \exists M' \in [M], \exists t \in X : M'[t]$.

A subset $X$ of transitions is qualified as live if and only if, it satisfies simultaneously the two following requirements:

1. We have to prove that it is always possible to fire a transition of $X$. For this, we have to show that every terminal SCC in the synchronisation graph contains at least one arc corresponding to the firing of a (synchronised) transition belonging to $X$, or it exists a node from which it is possible to fire a (internal) transition of $X$.

2. If the module can no more fire a synchronised transition (it reaches a Sync-closed SCC), we must prove that the Sync-closed SCC contains at least one arc corresponding to the firing of a transition belonging to $X$.

Proposition 5.1. $\forall s \in S, X \subseteq T_s$ is live $\iff$

$$\forall sec \in Term(SCC_{SG}) : X \cap Trans(sec) \neq \emptyset \lor \exists q \in sec : X \cap Trans([q]_s) \neq \emptyset$$

$\land \forall c_s \in SCC_s : c_s$ is a Sync-closed SCC $\Rightarrow X \cap Trans(c_s) \neq \emptyset$.

Proof:

$\Rightarrow$ Assume $X$ is live. This means that it is possible from any reachable marking to reach another marking in which one of the transitions of $X$ is enabled. In other words, for each terminal SCC $sec$ of the synchronisation graph either there exists a synchronised transition in $X$ belonging to $Trans(sec)$, or there exists a node $q$ in $sec$ from which a local transition of $X$ is enabled. Moreover, if a terminal SCC of the local state space is Sync-closed, then it has to enable a local transition of $X$. The proof of $\Leftarrow$ is similar to $\Rightarrow$.

Figure 7 gives a function which takes a set of transitions $X$ belonging to a module $s$ in order to decide whether it is live, or not. This function starts by marking the SCCs nodes in local graph which enable or contain a local transition of $X$, as well as their ancestors. Then it marks the nodes in the
Function CheckLiveness(Set of Transitions $X$, Module $s$): Boolean

forall the $c_s \in \text{Term}(SCC_s)$ do
  if $\text{Trans}(c_s) \cap X = \emptyset$ and $c_s$ is Sync-closed then return false

forall the $c_s \in SCC_s$ do
  if $\text{Trans}(c_s) \cap X \neq \emptyset$ then
    Mark $c_s$; Mark Ancs($c_s$)

forall the $M \notin c_s \in Q_{SG}$ do
  if ($M$ is marked) or ($\exists M', (-, f), (-) \in A_{SG}: f \cap X = \emptyset$) then Mark $M'$

if $\exists scc_{SG} \in \text{Term}(SCC_{SG}):$ scc$_{SG}$ is not marked then return false

return true

Figure 6. Checking the liveness of a set of transitions

synchronisation graph which are built from at least one marked SCC. The nodes of the terminal SCCs, of the synchronisation graph, enabling a synchronised transition of $X$ are also marked. If there exists a terminal SCC in the synchronisation graph without any marked node, we conclude that $X$ is not live. Otherwise, we check the second condition. For this, we consider all terminal SCCs of the module. For each of these, we check that either it contains a local transition of $X$ or it is not Sync-closed. If one node does not satisfy this requirement $X$ is not live, otherwise it is.

Example 5.1. We consider the local graph of module $A$ of Fig. 2 and the synchronisation graph of Fig. 5 reduced according to this module. Our aim is to check whether $X = \{ta; Sync2\}$ in module $A$ is live, or not. We begin by determining the Sync-closed SCCs of module $B$. There is one Sync-closed SCC, namely $A2$. This SCC does not contain any marking enabling a transition of $X$. Thus, $X$ is not live.

5.2. Deadlock local markings

The deadlock properties are interesting, since it is desirable to know whether a module can still be active or not, i.e. if it is always possible to find a path allowing a transition (local or synchronised) of the module to fire.

Definition 5.2. (Deadlock local marking)

Let $M_s \in Q_s$. $M_s$ is a deadlock local marking iff $\exists M \in [M_0]: \forall M' \in [M]: M'(t) \Rightarrow t \notin T_s$.

A local marking is a deadlock if and only if, it satisfies simultaneously the two following conditions:

1. The marking does not allow any transition to fire, then it belongs to a trivial Sync-closed SCC,
2. The marking allows only synchronised transitions of the other modules to fire, then it belongs to a trivial Sync-closed SCC.

Proposition 5.2. Let $SCC_{SG}$ be the set of SCCs of the synchronisation graph $Q_{SG}$.

$\forall s \in S, M_s \in Q_s$ is a deadlock local marking $\Leftrightarrow [M_s^c$ is a trivial SCC $] \land [M_s^c$ is a Sync-closed SCC $]$. 

Function \(\text{DetermineLocalDeadLock(Module } s)\): a Set of local markings

\[
\begin{align*}
\text{LocalDeadlock} & \leftarrow \emptyset \\
\text{forall the } M_s & \in Q \text{ s.t. } M'_s \in \text{Term}(SCC_s) \land \text{Trivial}(SCC_s) \text{ do} \\
\text{if } M'_s & \text{ is Sync-closed then} \\
\text{LocalDeadlock} & \leftarrow \text{LocalDeadlock} \cup \{M_s\}
\end{align*}
\]

return \(\text{LocalDeadlock}\)

Figure 7. Determine local deadlock markings of a module

Proof:
The proof of this proposition follows from the definition of a Sync-closed SCC. Since, in a Sync-closed SCC of a module \(s\), only local transitions are enabled. Further, a terminal trivial SCC does not enable any local transition. Thus, a trivial Sync-closed SCC corresponds to a local deadlock marking. \(\square\)

Figure 7 gives a function which returns the set of local deadlock markings of a module \(s\). The set \(\text{LocalDeadLock}\) will contain all local deadlock markings.

The idea is to determine terminal markings in the local graph. These markings are then those which do not allow any local transition to be fired. From these markings we consider only those which belong to Sync-closed SCCs. By this way, we get the local markings in which the module will stuck.

Here, we proceed by considering all local markings belonging to trivial and terminal SCCs of module \(s\). Among these, we add markings to the set \(\text{LocalDeadLock}\) which belong to Sync-closed SCCs.

Example 5.2. Here, we propose to determine the all local deadlock markings of module \(B\). For this, we consider the local graph of module \(B\) illustrated by Fig. 2 and its associated reduced synchronisation graph of in Fig. 5. The local graph shows that there exists one Sync-closed SCC, namely \(B2\). This SCC is trivial. Therefore, \(B2\) is a local deadlock marking of module \(B\). It means that module \(B\) stops evolving when it reaches the state \(B2\).

5.3. Home markings

Here, we propose to check whether a given set of local reachable markings in a module is a home space or not. A set of local markings \(X\) to module \(s\) is qualified as a home space if it is always possible from any local marking of module \(s\) to reach a local marking of \(X\).

Definition 5.3. Let \(X \subseteq Q_s\).

\(X\) is a home space iff \(\forall M_s \in [M_0]_s, \exists M'_s \in X: M'_s \in [M]_s\).

To determine whether a set of local markings \(X\) in a module \(s\) is a home space, we have to prove the two following requirements:

1. It is always possible to reach a marking of \(X\). For this, we have to show that all terminal SCCs of the synchronization graph have a node which contains a marking of \(X\) in its local successors.
2. If the module \(s\) can no more fire a synchronised transition (it reaches a Sync-closed SCC), we must show that at least one marking of \(X\) is always reachable using internal transitions only.
Function $\text{HomeSpace}(\text{Set of Markings } X): \text{Boolean}$

1. **forall the** $c_s \in \text{Term}(\text{SCC}_s)$ **do**
   - if $c_s \cap X = \emptyset \land c_s \text{ is Sync-closed} \text{ then return false}$
2. **forall the** $c_s \in \text{SCC}_s$ **do**
   - if $X \cap c_s \neq \emptyset$ then Mark $c_s$ and $\text{Anc}(c_s)$
3. **forall the** $M \neq c_s \in Q_{SG}$ **do**
   - if $M \cap c_s$ is a marked SCC then Mark the node $M$ in $Q_{SG}$
4. if $\exists \text{scc}_{SG} \in \text{Term}(\text{SCC}_{SG}) \text{ s.t. } \text{scc}_{SG} \text{ has no marked node}$ then return false

return true

Figure 8. Check whether a set of local markings is home space, or not.

**Proposition 5.3.** Let $\text{SCC}_{SG}$ be the set of SCCs of the synchronisation graph $SG$.

$\forall s \in S, X \subseteq Q_s$ is a home space $\iff [\forall \text{scc} \in \text{Term}(\text{SCC}_{SG}) : \exists q \in \text{scc}, X \cap [[q_s]] \neq \emptyset] \land [\forall c_s \in \text{SCC}_s : c_s \text{ is a Sync-closed SCC } \Rightarrow X \cap c_s \neq \emptyset]$.

**Proof:**

$\Rightarrow$ Assume $X$ is home space. This means that it is possible to reach a marking of $X$ from any reachable marking. Then, every terminal SCC of the synchronisation graph must contain a node $q$ from which any marking of $X$ is locally reachable. Further, every Sync-closed SCC must contain a marking of $X$. The proof of $\Leftarrow$ is similar to $\Rightarrow$. $\square$

Figure 8 gives a function which takes a set $X$ of local markings in module $s$. It returns the value $true$ if $X$ is home space, else the value $false$. It starts by marking the nodes of $X$ in the SCC graph of module $s$ as well as their ancestors. Then, it considers the nodes in the terminal SCCs of the synchronisation graph which are built only from marked SCCs. It marks those which contain at least one marking of $X$ in their local successors. If there exists a terminal SCC of the synchronisation graph without marked node, then the first part of the condition is not satisfied, thus $X$ is not home space. Otherwise, we have to check the second part of the condition. We consider terminal SCCs in local state space of the module. For each of these, we check that it is not a Sync-closed SCC, or that it contains a marking of $X$. If one node does not satisfy these two requirements, $X$ is not home space, otherwise it is.

**Example 5.3.** We consider the local graph of module $C$ of Fig. 2 and the reduced copy of synchronisation graph of Fig. 5 associated with this module. We apply the algorithm of Fig. 8 to check whether the set $X = \{C1, C2\}$ of local markings is home space, or not. We proceed by determining all Sync-closed SCCs in the local graph of module $C$. There are two Sync-closed SCCs, namely $C1$ and $C2$. Therefore, we continue the verification. We mark the nodes $C1$ and $C2$ in the local state space of module $C$. The nodes $A1B1C1$ and $A1B4C2$ in the synchronisation graph are then marked. Thus, $X$ is home space.

6. **Distributing a modular state space**

Considering a modular Petri net in a distributed environment, our aim is to allow a machine associated with a module to check its local properties without communicating with other machines, i.e. performing
local verification by applying the algorithms described in the latter section. For this, we propose to build the modular state space in a distributed manner, compute Sync-closed SCCs and performing the reduction of the synchronisation graph according to every module. Every machine has to maintain the local graph of its associated module and a reduced copy of the synchronisation graph. The local verification is then performed by exploring both graphs.

We do not give the algorithms performing the distributed building of a modular state space, since these algorithms can be easily derived from those proposed for timed modular Petri nets [14]. The latter paper focuses on the distributed building of a modular state space for a timed modular Petri net using a coordinator process and a set of worker processes. In our work, we follow that same architecture. We assume that a worker process can communicate only with the coordinator. The coordinator process is responsible for the building of the synchronisation graph. Each worker process is assigned to one module, and therefore it is responsible for the building of its local graph.

After the building of the distributed modular state space, such that the coordinator generates the synchronisation graph, and every worker process builds the local graph of its associated module, we propose to compute Sync-closed SCCs in a distributed manner. Then, according to each module a reduced copy of the synchronisation graph is built by the coordinator process. Every reduced copy is then sent to its correspondent worker process. As a final result, each worker process maintains a local graph of a module and a copy of the synchronisation graph reduced according to the same module.

Let us consider a distributed modular state space, such that the coordinator maintains the synchronisation graph and every worker process maintains the local graph of its associated module. We proceed by computing Sync-closed SCCs in a distributed manner. Figure 9 presents the algorithm to compute Sync-closed SCCs executed by the coordinator process. For every node \( q \) of the synchronisation graph, the coordinator process asks every worker process to send the terminal SCCs locally reachable from \( q \) by sending the message \textit{ASK}. When the terminal SCCs are received, the coordinator determines from them the set of Sync-closed SCCs. The determined sets of Sync-closed SCCs are sent back to worker processes. Figure 10 gives the algorithm to be executed by a worker process. A worker process executes a loop until the coordinator instructs to stop through the receipt of the message \textit{STOP}. When a message \textit{ASK} is received, the worker process computes and sends a set of terminal SCCs locally reachable from the specified SCC in the message. The Sync-closed SCCs are marked in the local graph when it is instructed by the coordinator.

Once Sync-closed SCCs of modules are computed, the coordinator performs the reduction of the synchronisation graph according to every module. Every reduced copy is then sent to its correspondent worker process. Finally, a worker process can perform the verification of local properties of its associated module without any further communication as it is described in the last section.

**Figure 9.** Algorithm for the coordinator process

```plaintext
for all \( q \in Q_{SG} \) do
\[ \forall s \in S, \text{SEND}(s, \text{ASK}_q s) \]
forall the \( s \in S \) do
\[ message \leftarrow \text{RECEIVE}(s) \]
\[ \text{SyncClosedSCC}_s \leftarrow \text{ComputeSyncClosedSCC}(q, s, \text{TermSCC}) \]
\[ \forall s \in S, \text{SEND}(s, \text{SCLOSED}_s \text{SyncClosedSCC}_s) \]
\[ \forall s \in S, \text{SEND}(s, \text{STOP}) \]
```
repeat
  \[message \leftarrow \text{RECEIVE}()\]
  if \(message \neq \text{STOP}\) then
    if \(message = \text{ASK} q_s\) then
      \(\text{TermSCC}_s \leftarrow \text{Term}(q_s) /\text{Term}(q_s)\) computes the terminal SCCs reachable from \(q_s\).
      \(\text{SEND}(\text{TERM TermSCC}_s)\)
    else
      \(\forall c_s \in \text{SyncClosedSCC}_s, \text{Mark} c_s\) as a Sync-closed SCC in \(Q_s\)
  until \(message = \text{STOP}\);

Figure 10. Algorithm for a worker process

Proof:
We start by proving the convergence of the proposed distributed algorithm.

The coordinator process initiates the determination of Sync-closed SCCs by sending the message "ASK". For every node of the synchronisation graph, it collects necessary information from worker processes in order to compute Sync-closed SCCs locally reachable from the node. Then, the results are transmitted to worker processes. When all nodes of the synchronisation graph are processed, the coordinator process send the message "STOP" to all worker processes in order to terminate them. Since the considered modular Petri net is assumed to be bounded, then its modular state space consists of finite graphs. Therefore, the algorithm terminates.

We prove now the correctness of the distributed algorithm. We have assumed that a worker process is associated with every module of the considered system, such that each worker process builds the local graph of its associated module. Further, we have assumed that there is an extra process called the coordinator process. The later has to build the synchronisation graph. By this way, the built graphs constitute the complete modular state space where each local graph of one module is maintained by one worker process, whereas the coordinator maintains the synchronisation graph. To determine Sync-closed SCCs, the coordinator starts by collecting the set of terminal SCCs locally reachable for every node of the synchronisation graph. Once, the coordinator has all terminal SCCs which are reachable from a node of the synchronisation graph, it performs the computation of Sync-closed SCCs by calling the function described in Figure 3. The latter function follows from propositions 3.1 and 3.2. Further, the computation of Sync-closed SCCs is performed for every node of synchronisation. This guarantees that all reachable terminal SCCs are checked. Thus, the proposed distributed is determines correctly the Sync-closed SCCs.

7. Conclusions and perspectives

In this paper, we have proposed an approach to check local properties of a system in a distributed environment. By considering a modular system modelled by a modular Petri net, we have proposed an approach allowing to check local properties of a module by exploring its local information and some global information. Indeed, we have adapted to use the modular state space instead of the ordinary state space. In
addition to its reduced size, allowing to tackle the explosion problem of state space, a modular state space simplifies the integration of a distributed verification approach. By building the modular state space in a distributed manner, each module is associated with one machine. A coordinator process is introduced to build the synchronisation graph, to determine the sync-closure property for terminal SCCs of modules and to coordinate the different machines during the building of the modular state space. Then, we have proposed a technique to reduce the synchronisation graph according to one module. Every reduced copy of the synchronisation graph associated with one module, is built by the coordinator and then sent to the machine maintaining the module. Hence, thanks to the property of Sync-closed SCCs, the verification of local properties of a given module is performed by exploring its local state space and its reduced copy of the synchronisation graph. This is advantageous, since it avoids communication between modules once the graphs are generated and distributed on different machines. As a perspective to our work, we find that it will be very interesting to perform directly the building of the reduced copies of synchronisation graph by avoiding the construction of the whole synchronisation graph. Further, we believe that it is possible to enhance the structure of modular state spaces in order to better support distributed approaches.

References


