http://www-lipn.univ-paris13.fr/~petrucci/PAPERS
In Proc. 4th Int. Conf. on Application of Concurrency to System Design (ACSD'2004), Hamilton, Canada, June 2004.
pages 185–194, IEEE Computer Society Press, 2004.

Modular Analysis of Systems Composed of Semiautonomous Subsystems *

Charles Lakos
Computer Science Department
University of Adelaide
ADELAIDE, SA 5005
AUSTRALIA
Charles.Lakos@adelaide.edu.au

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

Abstract

This paper reviews a proposal for the modular analysis of Petri nets and its applicability to factory automation systems. It presents new algorithms to harness this modular analysis in the determination of reachable states with specified partial markings, to determine possible deadlocks, both global and local, and also liveness. These algorithms have been implemented in a prototype tool which has then been used to solve a problem in factory automation which, even for relatively simple configurations, can lead to state spaces beyond the capabilities of many analysis tools.

1. Introduction

Factory automation systems commonly consist of semiautonomous subsystems. Thus, an automated manufacturing system will consist of a number of workstations which perform local operations, such as processing parts, and which interact periodically in some way. For example, a factory floor may have automated guided vehicles (AGVs) travelling between workstations and delivering parts and products [6]. Given the flexible nature of such systems, it is necessary to guarantee that they behave correctly. For example, it will be important to ensure that the AGVs do not collide on the factory floor, and that there is no resource contention leading to a deadlock either of the whole system or of a single workstation.

Of course, the problem here is not just to detect that collision or deadlock may occur, but also to ensure that it does not occur, by controlling the movement of the AGVs. The approach is to control the departure of the AGVs from the workstations, the input and completed parts stations. Once an AGV has left such a station, it progresses autonomously until it reaches the next one. The question then is how to evaluate the control policies or even to automate the design of such a policy so as to maximise the possible behaviour of the system while preventing collisions or deadlocks.

In order to explore these properties and control policies, the analysis of such systems can present a real challenge. Even for a simple factory floor configuration with three workstations, two input parts stations, one completed parts station and five AGVs, the problem is quite complex. Our Petri net model of this configuration has some 31 million states which may well exceed the capabilities of many analysis tools.

The earlier paper [6] proposed a very specific solution to the problem: off-line computations determined the paths between the control points and the possible collision points. These computations were only performed once as they identified the structure of the system. Then some on-line computations determined the maximally permissive control policies for individual states or markings.

Another attempt [9] used various reduction techniques [1] to reduce the state space to manageable proportions: agglomeration, removal of implicit places, etc. These techniques helped to reduce the state space by approximately two orders of magnitude to some 312,000 reachable states.

*This work was started when both authors were at LSV, CNRS UMR 8643, ENS de Cachan, France.

The above approaches were considered to be rather limited, either in requiring a very specific solution, or in the level of reduction achieved in the state space. Accordingly, we here present a more general approach based on the inherent modularity of the system. We have already observed that such factory control architectures are typified by a number of semiautonomous subsystems. These are similar to Cyclic Communicating Processes [11]. A modular analysis approach would examine in isolation, as far as possible, the local behaviour of each subsystem, and then separately consider the synchronisation between the subsystems. In this fashion, we avoid exploring the many possible interleavings of activity of the subsystems. As we shall see, this reduces the state space from some 31 million states to some 900 states.

In section 2, the paper recalls the definitions of modular state spaces from [2]. Actually, we present a slight optimisation which is used in our prototype implementation. In section 3, we introduce new algorithms for the construction of the modular state space, as well as for the verification of various properties. These algorithms have been implemented in a prototype tool, and section 4 reports on the application of this tool and these modular analysis techniques to the above problem of the automated factory floor. Finally, we present our conclusions in section 5.

2. Modular State Spaces

2.1. Definitions of Modular Petri Nets

We first recall the basic definitions and notations for Petri nets, their markings, enablings and occurrence rules:

Definition 1 *A* **Petri net** *is a tuple* $PN = (P, T, W, M_0)$*, 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* $\mathbb{N}$*, and* M_0 *is the* **initial marking***, namely a function mapping from* P *into* $\mathbb{N}$*.*

Definition 2 *A* **marking** *is a function* M *mapping from* P *into* $\mathbb{N}$*. The set of all markings is denoted by* $\mathbb{M}$*. A transition* t *is* **enabled** *in a marking* M*, denoted by* $M[t\rangle$*, iff* $\forall p \in P : W(p,t) \leq M(p)$*. When a transition* t *is enabled in a marking* M_1 *it may* **occur***, changing the marking* M_1 *to another marking* M_2*, defined by:* $\forall p \in P : M_2(p) = (M_1(p) - W(p,t)) + W(t,p)$*. The set of markings* **reachable** *from a marking* M*, is:* $[M\rangle = \{M' \mid \exists \sigma \in T^* : M[\sigma\rangle M'\}$.

Modular Petri nets are defined in a similar manner. In this paper, we consider only modules synchronised through shared transitions. This is relevant as the problems we tackle are synchronising semiautonomous systems. Therefore, we simplify the definitions of [2] which considered communication through places as well as transitions.

Definition 3 *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:* $\forall s_1, s_2 \in S : [s_1 \neq s_2 \Rightarrow (P_{s_1} \cup T_{s_1}) \cap (P_{s_2} \cup T_{s_2}) = \emptyset]$.
   - $P = \bigcup_{s \in S} P_s$ *and* $T = \bigcup_{s \in S} T_s$ *are the sets of all places and all transitions of all modules.*
2. $TF \subseteq 2^T \setminus \{\emptyset\}$ *is a finite set of non-empty* **transition fusion sets***.*

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

Definition 4 *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. $\forall p \in P, \forall tg \in TG$:

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

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 5 *A transition group tg is* **enabled** *in a marking M, denoted by $M[tg\rangle$, iff:*

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

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:*

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

Example Figure 1 depicts a modular PT-net consisting of three modules A, B and C. Modules A and B both contain transitions labelled F1 and F3, while modules B and C both contain transition F2. These matched transitions are assumed to form three transition fusion sets.

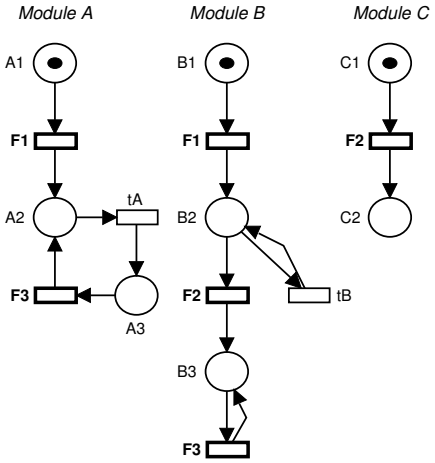

Figure 1. Modular PT-net with modules A, B and C.

2.2. Formal Definitions of State Spaces

In this section, we will recall the formal definitions of the modular state space introduced in [2].

2.2.1 State Spaces of Petri Nets

The state space (also named Occurrence Graph) of a Petri net is represented as a graph which contains a node for each reachable marking and an arc for each possible transition occurrence.

Definition 6 *Let $PN = (P, T, W, M_0)$, be a Petri net. The* **State Space** *of PN is the directed graph $SS = (V, A)$, where:*

1. *$V = [M_0\rangle$ is the set of vertices.*
2. *$A = \{(M_1, t, M_2) \in V \times T \times V \mid M_1[t\rangle M_2\}$ is the set of arcs.*

Example The (full) state space for the modular PT-net of figure 1 is shown in figure 2. Note that the initial state is shown as A1B1C1, thus indicating that place A1 is marked with a token in module A, place B1 is marked with a token in module B, and place C1 is marked with a token in module C. In this initial state, only transition F1 is enabled, its occurrence leading to state A2B2C1.

This simple example was chosen to illustrate the modular determination of deadlock and liveness (in section 3), rather than the possible reductions in the size of the state space.

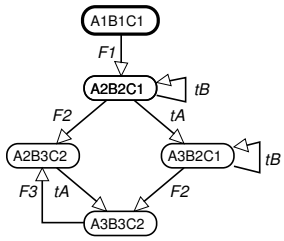

Figure 2. The full state space of the system.

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.

2.2.2 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 global marking M by occurrences of transitions in module s only.

We use $M_1[[\sigma\rangle\rangle M_2$ to denote that M_2 is reachable from M_1 by a sequence $\sigma \in (T \setminus TF)^* TF$ of internal transitions followed by a fused transition.

For any reachable marking M, we use $M^{\mathscr{q}}$ to denote the product (or tuple) of Strongly Connected Components (SCCs) M_s^c of the individual modules:

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

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

Definition 7 *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)* $V_s = \bigcup_{v\in(V_{SG})_s} [v\rangle_s$.
   - *(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)* $V_{SG} = [[M_0\rangle\rangle^{\phi} \cup \{M_0^{\phi}\}$.
   - *(b)* $A_{SG} = \{(M_1^{\phi}, (M_1'^{\phi}, tf), M_2^{\phi}) \in V_{SG} \times ([M_0\rangle^{\phi} \times TF) \times V_{SG} \mid M_1' \in [[M_1\rangle \wedge M_1'[tf\rangle M_2\}$.

Explanation
(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^{ϕ} 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 only contain 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.

Note that in [2], the arc labels of SG contain both the source and the destination state of a fused transition. Here, we just store the product of SCCs of the source state, which is sufficient for our purposes and optimises the state space arcs.

Example The modular state space for the modular PT-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).

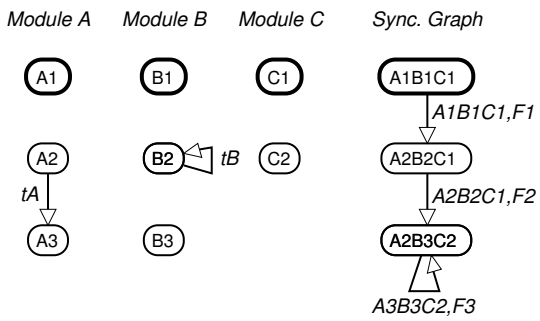

Figure 3. The modular state space.

3. Algorithms for Modular Petri Nets

Having reviewed the definitions of modular state spaces, we now present new algorithms for the construction of these state spaces and for the verification of various properties. Although sketches of algorithms were introduced in [2], we here give more detailed and optimized abstract algorithms which reflect the prototype tool implementation. Moreover, the algorithms are extended so as to be able to check not only global but also local properties.

3.1. Construction of the Modular State Space

The algorithm for the construction of the modular state space presented in [2] interspersed the computation of the local state space with the elaboration of the synchronisation graph.

In our prototype tool, we have implemented a different approach. The algorithm works in three main stages: firstly, the potential local state space for each module is computed, considering the module as standalone; secondly, the synchronisation graph is computed (thereby determining exactly which fused transitions can occur); and thirdly, the unreachable parts of the local state spaces are removed (given the global knowledge of which fused transitions are enabled).

This approach to computing the modular state space has the advantage of being able to be applied not only to Petri nets but also to other models of concurrency such as synchronised automata.

Algorithm

procedure MSS()
begin
/* Part 1 - compute local state spaces of modules */
for all $s_i \in S$ do
  $SS_i := state_space(s_i)$
  $SCC(SS_i \setminus TF)$; /* compute local SCCs */
endfor
/* Part 2 - compute the synchronisation graph */
$newwaiting := \emptyset$;
$Node(M_{01}^c, \ldots, M_{0n}^c)$;
repeat /* Process SG nodes */
  $waiting := newwaiting$;
  $newwaiting := \emptyset$;
  for all $v_{SG} \in waiting$ do
    /* all the unprocessed terminal nodes of SG */
    for all $i \in \{1..n\}$ do
      $Mark([[v_{SG}\rangle_i, SS_i)$
    endfor
    for all $tf \in TF$ do
      forall $M = (m_1, \ldots, m_n)$ s.t. $\forall i \in \{1..n\}$,
        $((tf \cap T_i \neq \emptyset \wedge m_i[tf\rangle_i) \vee (tf \cap T_i = \emptyset \wedge$
        $m_i^c = v_{SGi})) \wedge m_i$ is marked do
        $M' = (m'_1, \ldots, m'_n)$ s.t.
          $\forall i \in \{1..n\}$, if $m_i[tf\rangle_i$ then $m_i[tf\rangle_i m'_i$
                                  else $m'_i = m_i$
                endif
        $Node(M'^{\phi})$;
        $Arc(v_{SG}, (M^{\phi}, tf), M'^{\phi})$;
      endfor
    endfor
until $newwaiting = \emptyset$ /* all SG nodes processed */
/* Part 3 - remove local unreachable parts */
forall $i \in \{1..n\}$ do
  $Remove_Arcs(TF_i, SS_i)$;
  forall $v_{SG} \in V_{SG}$ do
    $Mark([[v_{SG}\rangle_i, SS_i)$;
  endfor
  $Remove_Unmarked(SS_i)$;
endfor
end

This algorithm also optimises the number of nodes in the synchronisation graph, as it does not represent equivalent occurrences of fused transitions having different markings for the modules which do not participate in the synchronisation. If this were not the case, then in our example a node A3B2C2 would have been constructed as the destination of an arc with source A2B2C1 and label (A3B2C1,F2).

3.2. Reachability

Having constructed the modular state space, we now wish to check properties of the system under consideration. First, we want to find whether a given marking M is reachable or not, simply by examining the modular state space. The *set of ancestors* of a local marking M_s in the state space graph of module s is the set of SCCs from which M_s can be reached, i.e. $\forall s \in S, \forall M_s \in V_s$: $anc_s(M_s) = \{M_i^c \mid M_i \in V_s \wedge M_s \in [M_i\rangle_s\}$.

We now express the reachability property:

Proposition 1 *[2]*

$$M \in [M_0\rangle \Leftrightarrow [(\forall s \in S : M_s \in V_s) \wedge ((\prod_{s \in S} anc_s(M_s) \cap V_{SG}) \neq \emptyset)].$$

Algorithm The process to check that the reachability of a marking M is easily implemented by first looking at the restrictions of M to the modules. If for one module s, M_s is not in SS_s, then M is not reachable. Otherwise, we check if there exists a node v in the synchronisation graph from which M is locally reachable. This can be done efficiently, using the information of the SCCs of the modules.

function Reachable(Marking M)
begin
for all $s \in S$ do /* check local parts */
  if $M_s \notin V_s$
    then **return(false)**
  endif
endfor
/* All the local parts of the marking exist,
   check global reachability */
for all $s \in S$ do /* in all modules, */
  $Mark(M_s^c)$
  $Mark_Ancestor_SCCs(M_s^c)$
endfor
if $\exists v = (M_1^c, \ldots, M_n^c) \in V_{SG}$ s.t.
      $\forall i \in \{1..n\}, M_i^c$ is marked
  then **return(true)**
  else **return(false)**
endif
end

Function $Mark_Ancestor_SCCs$ is a recursive function which starts from a given SCC, and flags

its immediate ancestors. It is then called for all those ancestors that were not previously flagged.

Example Let us apply this to the example presented in figures 1-3, to check the reachability of A2B2C2. Node A2 is in SS_A, B2 is in SS_B, and C2 is in SS_C. The only ancestor of A2 in SS_A is A2, the ancestor of B2 in SS_B is B2, and the ancestor of C2 in SS_C is C2. Thus the cross-products of ancestors are {A2B2C2} which does not occur in SG. Hence, the last condition of the proposition is not satisfied and A2B2C2 is not reachable.

Partial reachability is also handled by our tool implementation. This means that it is possible to check that a combination of markings in some modules is reachable whatever the marking in the other modules. In that case, the same algorithm is used, but all the nodes of the modules without a specified marking are marked as suitable. Hence, all SCCs of these modules are marked.

3.3. Deadlocks

We will now give the property used to find dead markings directly from the modular state space.

Proposition 2 *(adapted from [2])*

$M \in [M_0\rangle$ *is dead* $\Leftrightarrow [(\forall s \in S :$
$(M_s)^c \in Term(SCC_s) \cap Trivial(SCC_s))$
$\wedge(\forall (v_1, (M_1^{\phi}, tf), v_2) \in A_{SG} : M_1^{\phi} \neq M^{\phi})]$.

Algorithm The algorithm we have designed finds all reachable dead markings of the system.

We first mark all the terminal and trivial SCCs of the modules. If there exists a module without such a marked node, this module always allows some local behaviour. Hence, the system is deadlock-free. Otherwise, we construct the product of marked SCCs which contain all potential deadlocks. We check for each of these if it is effectively reachable. If this is not the case, it is deleted as it does not constitute a reachable deadlock. In the end, the set of remaining elements is the set of all reachable deadlocks of the system.

function Deadlocks()
begin
for all $s \in S$ do
  $Mark_Terminal_Trivial_SCCs(SS_s)$
  if no SCC is marked /* deadlock-free system */
    then **return(empty)**
  endif
endfor
/* possible deadlocks */
$deadlocks := Product_Of_Marked_SCCs$;
$deadlocks :=$
  $deadlocks \setminus EnableFused(deadlocks, A_{SG})$;
for all $M \in deadlocks$ do
  if $not(Reachable(M))$
    then $deadlocks := deadlocks \setminus \{M\}$
  endif
endfor
/* deadlocks contains all reachable deadlocks */
return(deadlocks)
end

Function $Mark_Terminal_Trivial_SCCs$ flags the terminal and trivial strongly connected components of a graph; function $Product_Of_Marked_SCCs$ explicitly constructs the product of all marked SCCs in the local state spaces; $EnableFused(deadlocks, A_{SG})$ returns the elements of $deadlocks$ which enable a fused transition. They are obtained from A_{SG} considering the markings only of the modules participating in the synchronisation, whatever the marking of the other modules.

Example We apply this algorithm to the example presented in figures 1-3, to find all reachable dead markings.
We first mark the terminal SCCs in the local state spaces SS_s, i.e. A1, A3, B1, B2, B3, C1 and C2. Then, we construct the cross-product of these and remove from the set obtained the representative of nodes enabling a fused transition. The resulting set is {A1B2C2,A1B3C1,A1B3C2, A3B1C1,A3B1C2,A3B2C2}. All the markings left in this set are unreachable. Hence, the set of reachable dead markings is empty.

Other deadlock properties are also interesting, particularly in the context of semiautonomous subsystems. On the one hand, it is desirable to know whether a module can still be active, i.e. if it is always possible to find a path allowing a transition of the module to fire. Such a property is equivalent to the liveness of the set of transitions in this particular module (see section 3.4). On the other hand, even if this property is satisfied, the transition that fires might not change the actual marking of the module. Thus, checking if once a certain marking is obtained in the module, it can never evolve is also an interesting property. This is achieved by a subtle modification of the deadlocks algorithm.

3.4. Liveness

The algorithm we will now introduce checks the liveness of a set of transitions.

Proposition 3 *(adapted from [2])*

$$
\begin{array}{l}
X \subseteq T \textit{ is live} \Leftrightarrow [\forall scc \in \mathit{Term}(SCC_{SG}) : \\
\quad (X \cap \mathit{Trans}(scc) \neq \emptyset \\
\quad\ \vee \exists v \in scc : X \cap \mathit{Trans}([[v\rangle) \neq \emptyset] \\
\wedge [\forall v \in V_{SG} : \forall M \in [[v\rangle : \\
\quad (\forall s \in S : M_s^c \in \mathit{Term}(SCC_s)) \Rightarrow \\
\quad (\exists s \in S : X \cap \mathit{Trans}(M_s^c) \neq \emptyset) \\
\quad \vee (\exists (v, (M_1^{\prime}, tf), v_2) \in A_{SG} : M_1 \in [[M\rangle\ )].
\end{array}
$$

Trans maps a strongly connected component or a set of markings to the set of transitions which occur in the labels of the arcs between markings of the set.

Algorithm In our implementation of the algorithm to check liveness of a set X of transitions, the set of transitions can be all the transitions or all those of a module or any other combination of transitions spread across the modules.

The algorithm is structured in two steps. The first one detects the problematic markings which do not enable any transition of X in a straightforward manner — the local component for each module belongs to a terminal strongly connected component which does not contain a transition of X, and the marking does not allow a fused transition from X to fire. These markings are stored in a set *Problematic* together with the synchronisation node from which they are locally reachable. Then the second step only operates on these, checking whether it is possible to reach another synchronisation graph node enabling a transition of X. When this is the case, the element is removed from the set. This loop is repeated until stability is achieved. Then, if the set is empty, set X is live otherwise it is not.

function Liveness()
begin
$\mathit{Problematic} := \emptyset$
for all $v \in V_{SG}$ do
  for all $s \in S$ do
    for all $c \in \mathit{Terminal_SCCs}([v_s\rangle_s)$ do
      if $\mathit{Trans}(c) \cap X = \emptyset$
        then $\mathit{Mark}(c)$
      endif
    endfor
  endfor
  for all $(M_1^c, \ldots, M_n^c) \in$ $\mathit{Product_Of_Marked_SCCs}$ do
    if $\nexists a = (v, ((M_1^c, \ldots, M_n^c), t), \ldots) \in A_{SG}$,
      $t \in X$
      then $\mathit{Problematic} := \mathit{Problematic} \cup \{(v, (M_1^c, \ldots, M_n^c))\}$
    endif
  endfor
endfor
repeat
  $\mathit{Unmark_SG_Nodes}$;
  $\mathit{Mark_SG_Nodes}(\mathit{Problematic})$;
  for all $(v, (M_1^c, \ldots, M_n^c)) \in \mathit{Problematic}$ do
    if $\exists a = (v, ((M_1^c, \ldots, M_n^c), \ldots), v') \in A_{SG}$,
      s.t. v' is not marked
      then $\mathit{Problematic} := \mathit{Problematic} \setminus \{(v, (M_1^c, \ldots, M_n^c))\}$
    endif
  endfor
until stability
if $\mathit{Problematic} = \emptyset$
  then **return(live)**
  else **return(not live)**
endif
end

Function $\mathit{Terminal_SCCs}$ determines the terminal strongly connected components of a graph made up of the set of local markings supplied as parameter; $\mathit{Unmark_SG_Nodes}$ unmarks all the nodes of the synchronisation graph (in preparation for the next iteration); $\mathit{Mark_SG_Nodes}$ marks all the nodes of the synchronisation graph that belong to the set given as parameter.

Example We apply this algorithm to the example presented in figures 1-3, to check that $X = T$ is not live. We first start with node A1B1C1 of SG. In the SS_s of modules A, B and C, we mark the terminal SCCs reachable from it, i.e. nodes A1, B1, and C1. There exists a transition starting from A1B1C1 in SG, hence we don't change *Problematic* but continue by processing another node in SG, e.g. A2B2C1. In the SS_s of modules A, B and C we mark nodes A3 and C1. Node B2 is not marked as transition tB is enabled from B2. The product of SCCs is empty, so we process the other node, A2B3C2. We mark A3, B3 and C2. A3B3C2 allows a transition to fire. Hence, there is no node in set $\mathit{Problematic}$ and $X = T$ is live.

Now, we apply the algorithm to check if $X = \{F2\}$ is live, and hence that module C is locally live. We first start with node A1B1C1 of SG. In the SS_s of modules A, B and C we mark nodes A1, B1, and C1. The transition starting from A1B1C1

in SG is not F2, hence we add (A1B1C1,A1B1C1) to set $Problematic$. We continue by processing another node in SG, e.g. A2B2C1. In the SS_s of modules A, B and C we mark nodes A3, B2 and C1. A3B2C1 allows us to fire F2. Then we process the other node, A2B3C2. We mark A3, B3 and C2. F2 is not enabled in A3B3C2. Hence, we add (A2B3C2,A3B3C2) to $Problematic$. Then, we start the second part of the algorithm. The SG nodes in set $Problematic$ are marked, i.e. A1B1C1 and A2B3C2. A2B2C1 is an unmarked successor of A1B1C1, thus (A1B1C1,A1B1C1) is removed from $Problematic$. A2B3C2 does not satisfy the condition. Thus $X = \{F2\}$ is not live. Since {F2} is the complete set of transitions for module C, this also means that node C2 is a local deadlock of module C.

4. Application to the AGVs Problem

This section considers the application of the tool implementing our algorithms to the Automated Guided Vehicles (AGVs) problem from [6].

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 Petri net of figure 4 models the system. The various stations appear on the edges of the net. The two input stations are the subsystems consisting of sets of places {I1, I2} and {I3, I4} (and their neighbouring transitions). The three workstations are captured by the subsystems with sets of places {W11, ..., W14}, {W21, ..., W24} and {W31, ..., W36}. The output station is the subsystem consisting of places {O1, O2}. The subsystems for the various AGVs are modelled by the central parts of the net. Thus vehicle A is captured by the places {A1, ..., A6} and commutes between input station 1 and workstation 1. Vehicle B is captured by the places {B1, ..., B14} and commutes between input station 2 and workstation 2. Vehicle D is captured by the places {D1, ..., D6} and commutes between workstations 1 and 3. Vehicle E is captured by places {E1, ..., E10} and commutes between workstations 2 and 3. Finally, vehicle F is captured by places {F1, ..., F8} and commutes between workstation 3 and the output station.

The grayed boxes represent dangerous zones, i.e. areas where the presence of multiple AGVs will lead to a collision. The factory floor, as shown, does not directly exhibit controls of the AGVs. However, it is intended that the filled

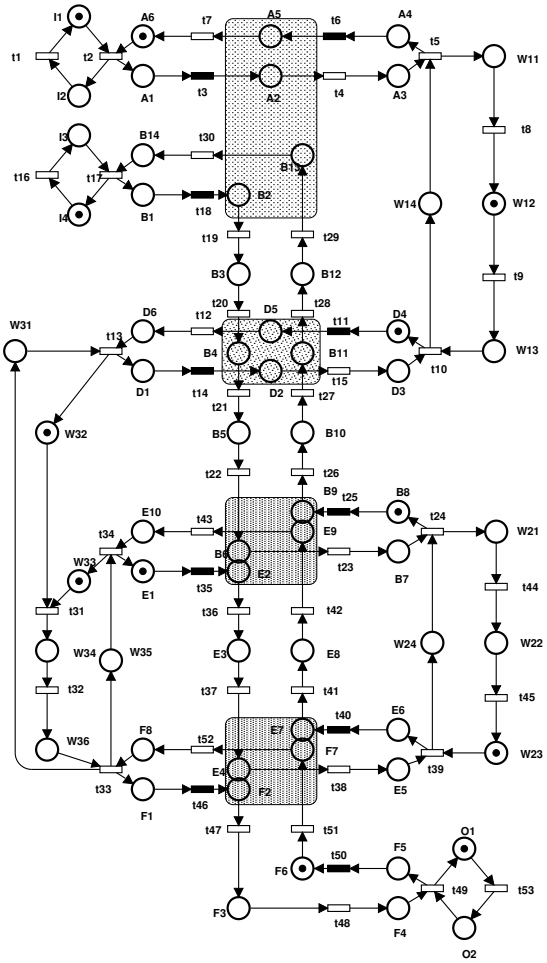

Figure 4. The five AGVs problem.

transitions represent possible control points. In other words, some controller can inhibit the firing of these transitions and thereby prevent collisions from occurring between the AGVs. The other transitions are not controllable, but can provide sensory information about the progress of the AGVs. It is then part of the problem to design the logic of the controller so as to eliminate the possibility of collisions, while minimising the disruption to the system. In other words, it is desirable to retain as much concurrent activity as possible, without allowing collisions to occur.

This example was addressed in [9], and could be solved using a combination of existing tools, namely DESIGN/CPN [7, 3] and HYTECH [5]. However, the state space explosion problem was immediately encountered, and reduction rules had to be applied, in order to achieve manageability.

It was shown in [9] that all states of the net in figure 1 can be reached, i.e. $30,965,760$ states.

4.1. Reachability of Forbidden States

The modular state space of the very same net contains a total of 900 nodes and $2,687$ arcs, computed in $30ms$[1]. The State Spaces of the modules contain a total of 64 nodes and 43 arcs altogether, while the synchronisation graph has 836 nodes and $2,644$ arcs.

The original approach to the problem was formulated in terms of reachability of forbidden states [6]. This approach can be emulated using modular analysis. Our tool allows us to specify partial markings, i.e. markings where the local states of only some modules are specified. Using the algorithm of section 3.2 it is possible to determine directly from the modular state space whether invalid situations specified by such partial markings can be reached. In this way, we can use the tool to determine that it is possible to reach a state where the first AGV is in A2 and the second in B13. In a similar fashion, it is possible to determine that all other 15 possible collisions are reachable.

4.2. Deadlock Detection

The issues to be addressed are not only the absence of collisions but also deadlock-freeness. Each AGV should not be stopped forever, even if the other ones still move. This does not correspond to a global deadlock problem, as presented in section 3.3, but to local deadlocks.

The question of local deadlocks for the AGVs problem can be resolved by checking that the set of local transitions for each module is live, using the algorithm of section 3.4. When applying this algorithm to the AGVs problem, we can determine that there is neither a global deadlock nor a local one, without ever having to examine the full state space. Therefore, the deadlock-freeness conditions are satisfied.

4.3. Preventing Collisions

Part of the AGVs problem (which has been solved above) is to identify the possible collision scenarios. The other part of the problem is to prevent these situations from occurring. It is for this reason that control transitions are included in the model (and identified by filled rectangles). A controller would be able to constrain the firing of these transitions if a collision were possible from the current state.

A simplified approach to the problem is possible by recognising that the control points occur immediately after synchronisation transitions. The original problem statement made this explicit by noting that the AGVs can be held at a station in order to avoid collisions. After leaving a station, the AGVs operate autonomously until they reach the next station.

Thus the problem of controlling the AGVs maps simply into the modular state space, since the nodes of the synchronisation graph determine the possible opportunities for restraining the AGVs. If we approach the problem in terms of reachability of forbidden states, then we can work backwards from such forbidden states to the node(s) of the synchronisation graph which lead to those states. If a given forbidden state can be reached from more than one node of the synchronisation graph, then the control logic needs to be applied to all such predecessors. Each of these predecessor nodes may be reached in the synchronisation graph by the firing of one or more synchronised transitions. If there is only one such synchronisation, then the associated AGV needs to be prevented from proceeding. If there are multiple synchronisations possible, then one of the AGVs involved will need to be inhibited.

4.4. More General Control Regimes

We have noted that synchronisation between AGVs and stations provides the points where the AGVs can be controlled to prevent collisions. This is ideal in linking the nodes of the synchronisation graph to the possible control points. Hence, all transitions to be controlled and the corresponding situation appear in the synchronisation graph just before the collision nodes. In other words, it is sufficient to work back from collision nodes to the preceding nodes of the synchronisation graph, determine whether this node arises from a synchronisation involving one of the collided vehicles, and then apply the control logic to this synchronisation.

It might be argued that such a fortuitous situation cannot be guaranteed. In other words, the control points may be distributed in a more irregular fashion. This more general situation can be addressed by exporting the control transitions. To do so, each control transition is made into a separate fusion set. Then its firing is recorded in the synchronisation graph, and we can work backwards

[1] The computations were done on a Pentium III, 1Ghz, with 512Mb of memory.

from the forbidden states or deadlock states to the synchronisation nodes which follow from the firing of such control transitions.

If we adopt this approach,the modular state space of the AGVs problem increases in size. When analysing the reachability of forbidden states (as in subsection 4.1), we end up with local synchronisation graphs containing a total of 64 nodes and 33 arcs, and a synchronisation graph of $23,120$ nodes and $93,952$ arcs.

There is a significant increase in the size of the synchronisation graph due to the fact that the controllable transitions are now explicitly shown. Nevertheless, graphs of some $30,000$ nodes can still be analysed using commonly available tools such as FSM [4], which was not the case with the original state space of some 31 million nodes.

5. Conclusions

This paper has reviewed a proposal for the modular analysis of Petri nets. It has presented new algorithms to harness this modular analysis in order to determine the reachability of specified partial markings, possible deadlocks, and whether a set of transitions is live. These algorithms have been implemented in a prototype tool which has then been used to solve a problem in factory automation which, even for relatively simple configurations, can lead to state spaces which are beyond the capabilities of many analysis tools.

This problem is particularly suited to modular analysis because it consists of a number of semiautonomous subsystems. These subsystems have significant local behaviour, occasionally interrupted by synchronisation with other subsystems. The modular analysis can explore the local behaviour without considering all the possible interleavings with the local behaviour of other subsystems.

The analysis of the factory automation problem was further facilitated by the observation that the nodes of the synchronisation graph determine exactly those points where control logic can be applied. Thus the modular analysis has significant benefits not just for the determination of potential problem states, but also by identifying the points where control logic can be applied to prevent those problem states from occurring.

In the case of a factory floor with two input stations, three workstations, one output station and five autonomous guided vehicles (AGVs), modular analysis reduced the size of the state space by over four orders of magnitude, thus making it amenable to analysis by a prototype tool. If a more flexible control environment were proposed, in which case the control transitions would need to be exported so as to appear explicitly in the synchronisation graph, then the benefits are less marked — some three orders of magnitude. By contrast, an earlier approach to the problem which applied net reduction techniques [9] achieved only two orders of magnitude reduction in the size of the state space. The modular approach has been compared to partial order reduction techniques in [10], which did not succeed for the AGVs problem.

Modular analysis is thus particularly suited to systems with strong cohesion and weak coupling, the desirable attributes of any modular system [8]. Where a system exhibits strong coupling between subsystems, the benefits are less marked.

References

[1] G. Berthelot. Checking properties of nets using transformations. In *Advances in Petri Nets*, volume 222 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 1985.

[2] S. Christensen and L. Petrucci. Modular analysis of Petri nets. *The Computer Journal*, 43(3):224–242, 2000.

[3] DESIGN/CPN *online*. `http://www.daimi.au.dk/designCPN`.

[4] *FSM library – General purpose finite-state machine software tools*. `http://www.research.att.com/sw/tools/fsm`.

[5] *A user guide to* HYTECH. `http://www.eecs.berkeley.edu/~tah/HyTech`.

[6] B. Krogh and L. Holloway. Synthesis of feedback control logic for discrete manufacturing systems. *Automatica*, 27(4), 1991.

[7] META Software and Aarhus University. *Design/CPN 3.0*, 1996. Also available as: `http://www.daimi.au.dk/designCPN`.

[8] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall Int., New York, 2nd edition, 1997.

[9] L. Petrucci. Design and validation of a controller. In *Proc. 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI'2000), Orlando, FL, USA, July 2000*, volume VIII, pages 684–688. International Institute of Informatics and Systemics, 2000.

[10] L. Petrucci. Modélisation, vérification et applications. Mémoire d'habilitation à diriger des recherches, Université d'Evry, Dec. 2002.

[11] P. S. Thiagarajan. Cyclic communicating processes. In *Proc. 3rd Int. Conf. on Application of Concurrency to System Design (ACSD'03), Guimarães, Portugal, June 2003*, page 4. IEEE Comp. Soc. Press, 2003.