Synthesis of Coloured Petri Nets from Natural-like Language Descriptions

Enrique Arjona[1], Graciela Bueno[1] and Ernesto López-Mellado[2]
[1]*Colegio de Postgraduados Campus Montecillo*
[2]*CINVESTAV-IPN Unidad Guadalajara*
México

1. Introduction

Coloured Petri nets (CPN) (Jensen, 1981) have been widely used for the modelling of tasks in flexible manufacturing systems at different levels of functioning (Aized et al., 2007; Da Silva et al., 2008; Diaz, 2009). Since their conception, there have been several attempts to take advantage of the formalism of CPN to adapt them as a universal, discrete-event modelling language by extending their original definition (Jensen, 1991; Yeung et al., 1999; MengChu, 2009). Despite these attempts, however, complex and large-scale models are very hard to build using CPN or their extensions.
In order to ease the modelling of complex systems using Petri nets, some synthesis methods have been proposed, both for ordinary Petri nets (Der Jeng & Di Cesare, 1990; Zhou et al., 1992; He et al., 2000; Badouel & Darondeau, 2004; Zhi-Jun et al., 2008; etc.) and for CPN (Micovsky et al., 1990; Baldassari & Bruno, 1991; Ezpeleta, 1993; Shang et al., 2004; Khadka, 2007; etc.). These methods can be classified into two groups.
The first group (Der Jeng & Di Cesare, 1990; Zhou et al., 1992; Ezpeleta, 1993; He et al., 2000; Badouel & Darondeau, 2004; Khadka, 2007) includes those methods that preserve the formal nature of Petri nets. In order to achieve this, the methods impose restrictions on the type of situations that can be modelled and do not include the indiscriminate use of shared resources and complex ordering and selecting criteria (AND's, OR's, NOT's, and their combinations). Therefore, the range of applicability of the methods is very limited. Der Jeng & Di Cesare (1990) review some representative methods, all of them are of low level and do not allow the modelling of shared resources or the modelling of ordering and selecting criteria. Zhou et al. (1992) present a method that allows the modelling of some particular types of shared resources but does not allow the modelling of ordering and selecting criteria. Ezpeleta (1993) includes a method for the synthesis of models expressed by a restricted class of CPN called simple sequential processes; in this method, ordering and selecting criteria are limited to the use of FIFO policies. The other methods included in this group use an interface to facilitate the modeling of the systems and a fixed catalogue of subnets for the synthesis of the corresponding Petri net; in none of these methods is allowed the modelling of ordering and selecting criteria. He et al. (2000) define manufacturing processes in an interface called IDEF3 (Integrated Definition 3) and transform the model obtained to a Petri net using a sequential cluster identification algorithm. Badouel & Darondeau (2004) establish relations that the system to model has with a predefined set of

paths and use a bijection to construct a Petri net of the system. Khadka (2007) uses a sequence chart to represent the system and transforms the chart to a CPN using a tool called LSCTOCPN (Live Sequence Charts to CPN).

The second group (Micovsky et al., 1990; Baldassari & Bruno, 1991; Shang et al., 2004; Zhi-Jun et al., 2008) comprises those methods that do not preserve the nature of Petri nets. They use Petri net extensions that include facilities of procedural nature that allow great flexibility with respect to the situations that can be modelled, but the models obtained cannot be formally analyzed because these models are hybrids composed of subnets and/or computer procedures. Micovsky et al. (1990) propose a method that uses modified CPN that are interpreted using a proprietary language called DOOR that is implemented in a TPL (typeless procedural language); the programs obtained are used as input of a simulator. Baldassari & Bruno (1991) present a method that obtains computer programs to the situations that can be modelled, but the models obtained cannot be formally analyzed because these models are hybrids composed of modified CPN subnets called PROT nets and computer object-oriented concepts. Shang et al. (2004) synthesize system behavioural specifications into a mixture of labeled Petri nets and CPN. Finally, Zhi-Jun et al. (2008) present a method to combine existing web service models and synthesize them into a mixture of control flow nets and Petri nets.

This chapter presents a method of synthesis of CPN for the formal specification of simple and complex tasks in flexible manufacturing systems. The method differs significantly from other published methods in that it preserves the formalism of CPN without imposing restrictions on the system modelled, and therefore it allows the modelling of shared resources and complex ordering and selecting criteria. The proposed method allows one to systematically obtain CPN models from declarative descriptions of a very high level of abstraction. These descriptions are activity models (ABAM) expressed in a natural-like language. The language is not extensive and it is easy to master. In the language, the stochastic occurrence of many events is implicitly embedded in exogenous and endogenous variable conditions, and material and resource flows can be easily modelled using a single flow statement. ABAM are built using the activity-based approach, a non formal modelling tool for discrete event systems, which has proven to be valuable for the modelling of complex systems and also to be user friendly, to such an extent that some event- and process-based simulation languages have input interfaces that use this approach.

The CPN generated can be structurally validated, analyzed, and used for control or simulation.

The remainder of the chapter is divided in 6 sections and an appendix. Section 2 is a brief account of similarities and differences between ABAM and CPN. Section 3 presents basic support modules for the CPN synthesis and the mathematical proofs of their validity. Section 4 discusses the main features of the natural-like language. Section 5 gives an outline of the proposed method to build CPN. Section 6 illustrates the synthesis method through an example of an automotive workshop, its complete specification in the natural-like language and the CPN synthesized. Section 7 discusses some conclusions. Finally, an appendix includes the formal grammar of the language.

2. ABAM and CPN

Formally, both ABAM and CPN can be defined as directed bipartite graphs (Jensen, 1981; Kreutzer, 1986) that include two kinds of nodes (ABAM have activities and waiting lines, CPN have places and transitions), sets of objects associated to those nodes (admissible entities in the case of ABAM, colours in the case CPN), input and output functions, and time functions (these are included only in temporized models). Pictorially, nodes are represented with circles and rectangles and functions as directed arcs that have associated symbolic expressions (see figs. 1 and 4).

At first glance it may appear that by associating waiting lines to places and activities to transitions ABAM and CPN are equivalent, nevertheless there are fundamental differences between them. (1) The expressive power (level of abstraction) of CPN is lower than that of ABAM because the former does not allow all kinds of input and output functions. (2) Transitions in CPN may represent events or activities, depending on the interpretation of the model. (3) In CPN there is no explicit association of objects to colours. (4) When the CPN definition includes time, temporization can be carried out indistinctly in the places or in the transitions. (5) Input and output functions in CPN are always expressed in mathematical and not in declarative form (despite the fact that functions in a CPN can be temporarily stated in declarative form, they must be able to be expressed as linear functions).

3. Modules for CPN synthesis

As was stated before, CPN definition does not allow for direct specification of all kinds of input and output functions. To facilitate the building of CPN, for systems where complex situations arise, a bottom-up synthesis process is proposed. CPN models are built using a set of CPN predefined modules, each one of them representing a particular situation. Predefined modules can be embedded between them to represent two or more particular situations simultaneously.

To determine a partition of the situations that can arise when modelling discrete event systems with CPN, and thus define the necessary modules for the synthesis, we performed an analysis of the models that can be obtained using the activity-based approach. This does not impose any restrictions because if on the one hand we cannot assume that in general ABAM are CPN, the reverse is true. Complex situations in CPN arise when we try to associate ordering criteria to places and individual or multiple selecting criteria to input-output functions of transitions (for example, when we want to include in a CPN an input function that has priorities in the selection of colours, and the number of these is not fixed but variable).

In total, 13 CPN modules were defined for the synthesis process, and their validity was proved using induction. Two of the simplest modules are the CPN that correspond to a FIFO and a LIFO waiting lines. Their pictorial representations are given in figs. 1 and 2.

Figure 1 depicts a CPN module that simulates a FIFO ordering of entities in a place P that has a capacity of n. Transitions TIN and TOUT are used to store/remove entities in/from the place. Transition TMOVE is used to move entities in the place. The colour sets used in the module are:

$E = \{ <e,i> ; i=1,\ldots,m \}$
$S = \{ <s,j> ; j=1,\ldots,n \}$
$ES = E \times S$

Colour set E represents the entity types that can be admitted in the place, colour set S represents numbered spaces in the place, and colour set ES represents entities stored in numbered spaces. The colour sets associated to the place and the transitions are:

$C(P) = ES \cup S$
$C(TIN) = C(TOUT) = E$
$C(TMOVE) = ES - \{ <<e,i>,<s,1>> ; i=1,...,m \}$

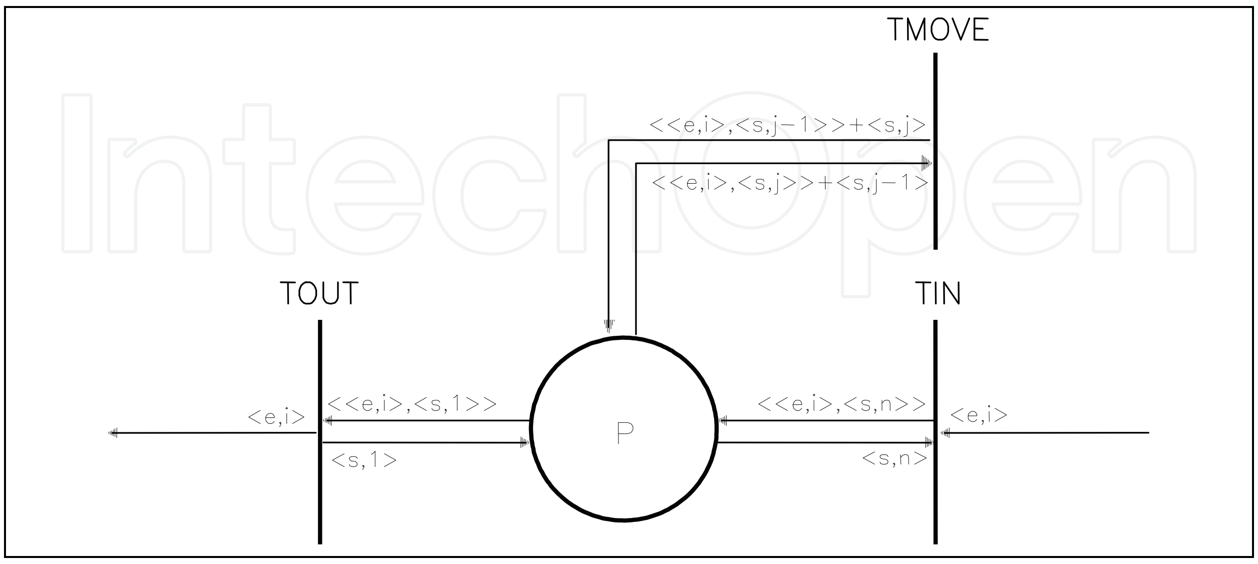

Fig. 1. A CPN module for FIFO ordering of entities in a place

The input/output functions of the transitions are given in Figure 1. Place P must be initialized with a set of n tokens (colour instances) taken from S and ES. All the elements of S that appear in these tokens must be different. That is, all the spaces of the place, occupied or unoccupied, must be included in the initialization. At any moment, the cardinality of the set of tokens in place P is equal to the maximum length of the waiting line.
The module execution is as follows: transition TIN stores an entity in the last position of the place (space n). Transition TMOVE moves an entity from its actual position to the previous one whenever possible. Transition TOUT removes the entity stored in the first position of the place.
A proof of the proper execution of the module using mathematical induction is the following: Assume that the place contains k (k<n) ordered entities according to a FIFO criterion, and that an entity arrives and is not properly ordered. This means that the entity was moved by transition TMOVE from the position n to a position p such that p>k+1 or p<=k. In the first case, transition TMOVE ceased firing in spite that the position p-1 was empty (by the induction hypothesis only the first k positions were occupied and p-1>k). In the second case, transition TMOVE moved the arriving entity to an occupied position (by the induction hypothesis first k positions were occupied and p<=k). Both cases lead to contradictions because transition TMOVE moves an entity if, and only if, the previous position is empty.
Figure 2 depicts a CPN module that simulates a LIFO ordering of entities in a place P that has a capacity of n. Transitions TIN and TOUT are used to store/remove entities in/from the place. Transition TMOVE is used to move entities in the place. The colour sets used in the module are:

$E = \{ <e,i> ; i=1,...,m \}$
$S = \{ <s,j> ; j=1,...,n \}$
$DS= \{ <s,n+1> \}$
$ES = E \times S$

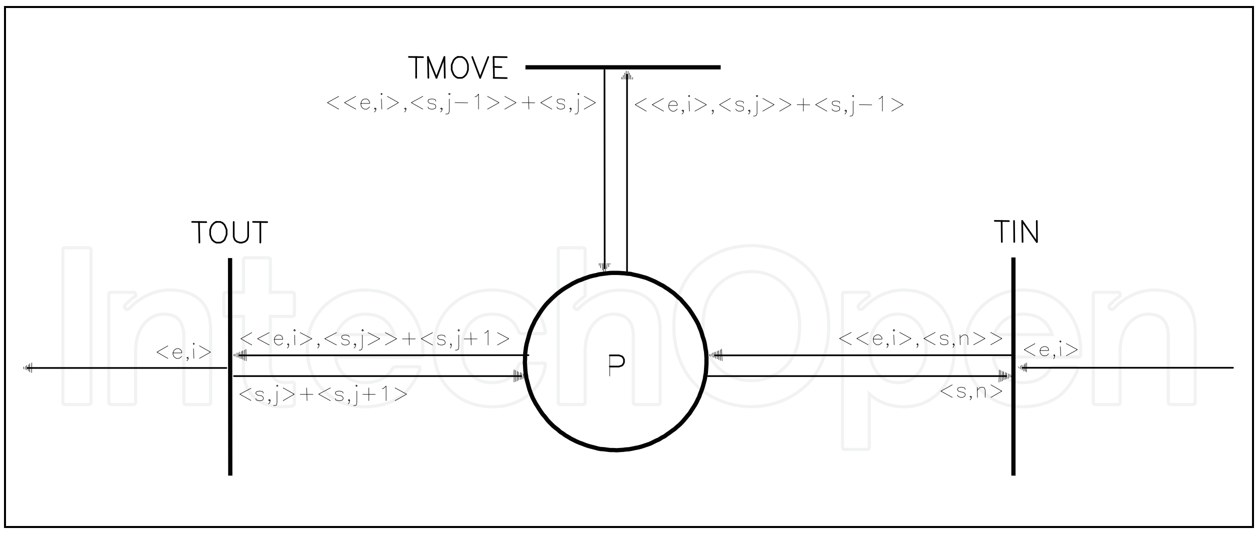

Fig. 2. A CPN module for LIFO ordering of entities in a place

Colour set E represents the entity types that can be admitted in the place, colour set S represents numbered spaces in the place, and colour set ES represents entities stored in numbered spaces. Colour set DS represents a dummy extra space used by transition TOUT when the place is full. The colour sets associated to the place and the transitions are:

C(P) = ES ∪ S ∪ DS
C(TIN) = C(TOUT) = E
C(TMOVE) = ES - { <<e,i>,<s,1>> ; i=1,...,m }

The input/output functions of the transitions are given in Figure 2. Place P must be initialized with a set of n+1 tokens. One token is taken from DS. The other n tokens are taken from S and ES. All the elements of S that appear in these n tokens must be different. That is, all the spaces of the place, including the dummy space, must be included in the initialization. At any moment, the cardinality of the set of tokens in place P is equal to the maximum length of the waiting line plus one.
The module execution is as follows: Transition TIN stores an entity in the last position of the place (space n). Transition TMOVE moves an entity from its actual position to the previous one whenever possible. Transition TOUT takes the entity stored in the last non-empty position of the place. This entity is easily recognized because the position next to it is always empty, even when the place is full. In this case, the empty position is the dummy space. Note that priority of transition TMOVE must be higher than the priority of transition TOUT. The proof of the proper execution of the module is similar to the proof given for the FIFO module.
The other modules deal with more complex situations, namely ascendant/descendent ordering criteria with respect to a given attribute, extraction of m items, contained from position i, of a n-position waiting line ($0 <= i <= n - m; m <= n; m$ predefined or variable), and accessing of a specific position of a FIFO/LIFO waiting line. In (Arjona & Bueno, 2007) are included two of the CPN modules for the modelling of complex selecting criteria (the selection of a constant and a variable number of entities from a set of places), mathematical proofs of their validity, and an example of their application to a real life simulation model of a sugarcane plantation.

4. Specification of models in a natural-like language

The specification of a model in the natural-like language consists of definition of entities, waiting lines, variables, and activities. Definition of entities and variables can include attributes, and definition of waiting lines can include ordering criteria. Activity definitions consist of the specification of starting and ending conditions, flows, and attribute modifications. Besides declarations, the language only uses two types of statements: the first one is for flow specifications and the second one for explicit state change specifications when events occur. These statements can model all individual and multiple flows mentioned in the preceding section.

Although the purpose of the natural-like language developed was to model flexible manufacturing systems, as a matter of fact, it is domain independent and was designed taking into consideration all kinds of possible material and resource flows in discrete event systems. This proved to be very useful for modelling very complex situations in real systems. In the appendix it is included the complete formal grammar of the language with an explanation of the symbols used to define it. In section 6 it is included an example of the language usage.

5. CPN synthesis from models specified in a natural-like language

The CPN modules and the natural-like language mentioned above have been utilized as the basis of a method for CPN synthesis. In this method, instead of using a bottom-up approach as before, a top-down technique is utilized. The CPN are synthesized by successive refinements of analysis of ABAM specified in the language.

The CPN synthesis process comprises six steps. The first five steps do not take into consideration complex situations in the input model and determine successively the place set, transition set, place set, transition colour set, and input and output functions. The sixth step performs refinements to the CPN already obtained in order to take into account complex situations.

Step 1. *Determination of the Place Set.* For the determination of the place set, entities and their attribute values in the input model are analyzed. There will be as many places as different states of the entities are actually used in the activities of the model. This avoids, to the extent possible, combinatorial explosion of the number of places and comprises a reduction process that unifies states, makes implicit some information (information that is not fundamental for the real-time execution of the model), and eliminates superfluous information (entities and attribute values that were defined but never used in the activities).

In the reduction process, entities and attributes are classified internally according to their use in the input model. Entities can be individual or multiple, depending on whether they represent a single resource of the system or a set of resources that have some logical or physical association. Attributes can be of constant or variable value. Attributes of constant value distinguish only members of the same family of resources and never represent the state of an entity. Attributes of variable value can be used to indicate contents, position, or physical characteristics. Attributes that indicate contents, independently of their values, can represent only two states of the entity to which they belong. Other attributes of variable value can represent as many states as values of the former are used in the activities. The number of places corresponding to an entity will be the product of the number of states that its attributes represent. When an entity has no attributes, it will be modelled with one place that represents its availability.

Step 2. *Determination of the Transition Set.* Transitions in a CPN correspond to state changes, and therefore there will be one transition in the transition set for each possible configuration of the system that allows the start of an activity. The number of input places of the transition will be the number of concurrent entities that the activity requires to start. The number of output places of the transition will be the number of entities that go out from the activity when it ends.

Step 3. *Determination of the Place Colour Set.* Because places represent a state of an entity, the colour set associated to a place will take into consideration only attributes of constant value, or of variable value that indicate contents and whose values are not uniquely represented by the state of the entity. The colour set consists of n-tuples with as many components as attributes of these types the entity has defined.

Step 4. *Determination of the Transition Colour Set.* The number of ways in which a transition can be fired depends on the different colours admissible in its input places. Therefore, the colour set associated to a transition will be the set of the n first natural numbers where n is the maximum of the cardinalities of the colour sets of its input places.

Step 5. *Determination of the Input and Output Functions.* Input (output) functions of a transition are the sum of individual functions associated to its input (output) places. The individual function of a place is the composition of two functions: the first one maps the set of colours of the place with the set of colours of the transition, and the second one determines which colours of the place are taken for each colour of the transition.

Step 6. *Refinements to Include Complex Situations.* In the last step of the synthesis process, complex situations in the input model are analyzed and the CPN already obtained is refined by introducing, one by one, the necessary modules. These refinements do not change the basic properties of the net, and its analysis can be done before this step is performed.

6. An example of CPN synthesis

Next, an example of the application of the synthesis process is included; it is taken from (Colom et al., 1990), where a CPN model is presented. The system considered is a puttying car body workshop designed and operated by a European automobile firm, whose layout is depicted in fig. 3.

The workshop has 12 working posts divided into two groups of 6. There is a conveying system that consists of roller tables where each table can contain one car body. Roller tables RT only convey. Roller tables TT in front of stations, in addition to conveying, rotate for loading and unloading. Roller tables ST, in addition to conveying, can also slide and allow distribution between groups of tables. A working post consists of three tables, one for loading (LT), one for processing (puttying) (PT), and one for unloading (UT). Loading and unloading tables in each workstation are integrated in a sliding bench that changes position for loading and unloading. At any moment a workstation can contain at most two car bodies. Car bodies in the conveying system have assigned the workstation number where they will be processed.

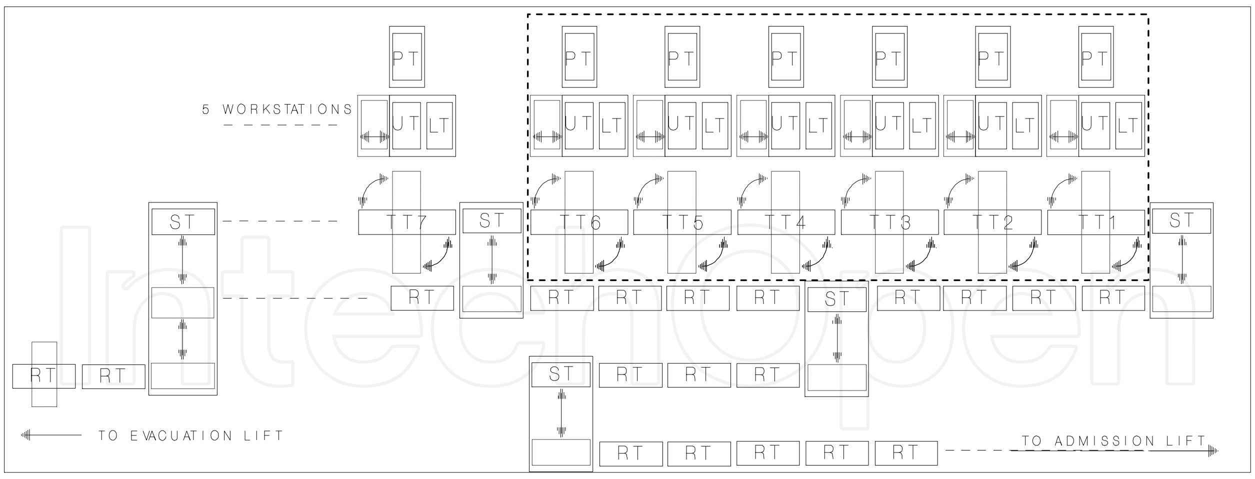

Fig. 3. Workshop layout

In this example, we consider only the section in the dashed rectangle. In general, the model will work as follows: car bodies arrive at the section on roller table number 1. Upon arrival, a pass with a number that corresponds to the workstation where it will be processed is assigned to the car body (in the complete model, passes are assigned elsewhere outside the section). Car bodies are moved from one roller table to the next until they arrive at the workstation to which they have been assigned. Upon arrival, they are transferred successively to the loading, processing, and unloading tables of the workstation, and again to the roller table in front of the workstation. After this, the car bodies will be moved from one roller table to the next until they arrive at roller table number 6, from where they leave the section.

An activity-based approach input model is depicted in fig. 4; it consists of three entities, three waiting lines, and seven activities. Entities considered are: passes (PASS AV), with one attribute that represents their number (its admissible values are integers from 1 to 6); roller tables (RT) with two attributes, the first represents its number (its admissible values are integers from 1 to 6) and the second the workstation number where the car body on the roller table will be or was processed and that we will call body destination (its admissible values are integers from -6 to 6, zero indicates that a roller table is free and a negative number that the body has already been processed); and workstations (WS) with four attributes, the first representing its number (its admissible values are integers from 1 to 6) and the others the states of its loading, processing, and unloading tables (its admissible values are integers 0 or 1). The bodies were not considered entities because the passes are sufficient to represent them. Also, the numbers of the passes were not considered in the attributes of the workstations because, as was stated above, pass numbers always coincide with the number of the workstations where the bodies are processed.

The waiting lines correspond to the inactive states of the entities and are: available passes, inactive roller tables, and workstation pool.

The activities are: body arrival (BA), body conveyance (BC), load from roller table (LFRT), load from loading table (LFLT), unload from processing table (UFPT), unload from unloading table (UFUT), and body leaving (BL). Following it is given the complete description of the model in the natural-like language.

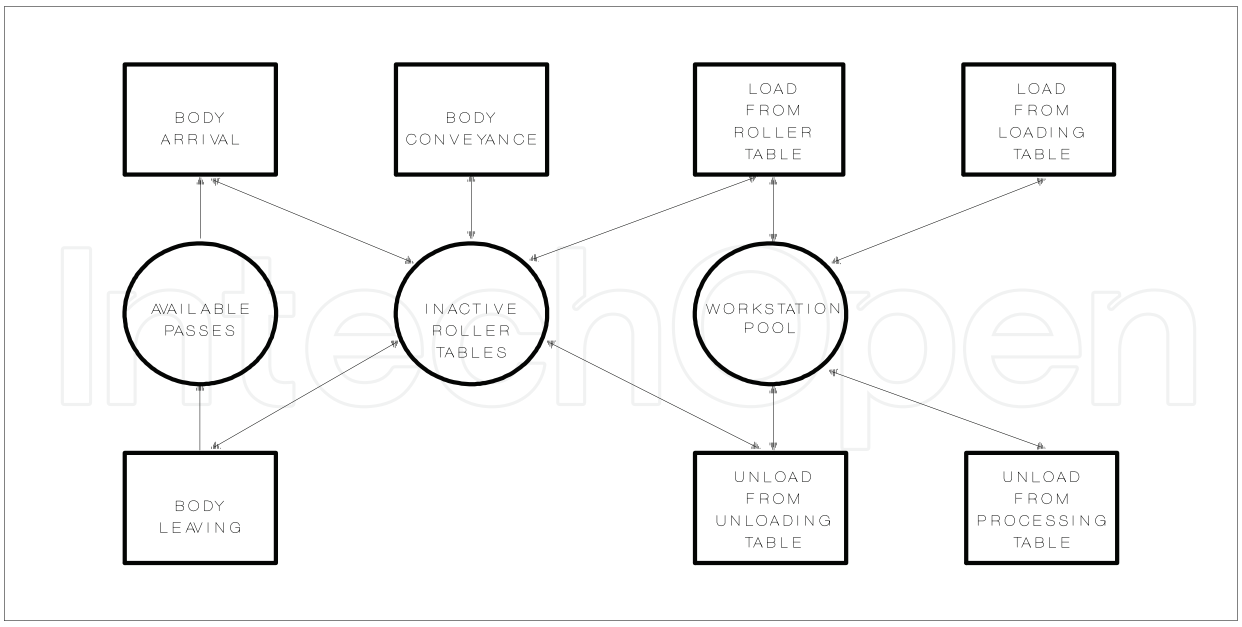

Fig. 4. Activity model of the workshop

```
MODEL NAME:
   AUTOMOTIVE_WORKSHOP.
ENTITIES:
   PASS WITH THE FOLLOWING ATTRIBUTES:
      NUMBER (ADMISSIBLE VALUES 1 TO 6).
   ROLLER_TABLE WITH THE FOLLOWING ATTRIBUTES:
      NUMBER (ADMISSIBLE VALUES 1 TO 6)
      BODY_DESTINATION (ADMISSIBLE VALUES -6 TO 6).
   WORKSTATION WITH THE FOLLOWING ATTRIBUTES:
      NUMBER (ADMISSIBLE VALUES 1 TO 6)
      STATE_OF_LOADING_TABLE (ADMISSIBLE VALUES 0 TO 1)
      STATE_OF_PROCESSING_TABLE (ADMISSIBLE VALUES 0 TO 1)
      STATE_OF_UNLOADING_TABLE (ADMISSIBLE VALUES 0 TO 1).
WAITING LINES:
   AVAILABLE_PASSES CONTAINS PASSES.
   INACTIVE_ROLLER_TABLES CONTAINS ROLLER TABLES.
   WORKSTATION_POOL CONTAINS WORKSTATIONS.
EXTERNAL VARIABLES:
   START_OF_BODY_CONVEYANCE.
   END_OF_BODY_CONVEYANCE.
   START_OF_LOAD_FROM_ROLLER_TABLE.
   END_OF_LOAD_FROM_ROLLER_TABLE.
   START_OF_LOAD_FROM_LOADING_TABLE.
   END_OF_LOAD_FROM_LOADING_TABLE
   START_OF_UNLOAD_FROM_PROCESSING_TABLE.
   END_OF_ UNLOAD_FROM_PROCESSING_TABLE.
   START_OF_UNLOAD_FROM_UNLOADING_TABLE.
   END_OF_UNLOAD_FROM_UNLOADING_TABLE.
```

```
    START_OF_BODY_LEAVING.
    END_OF_BODY_LEAVING.
ACTIVITY:
    BODY_ARRIVAL.
  ENTITY FLOWS:
    GET A PASS FROM AVAILABLE_PASSES.
    GET A ROLLER_TABLE WITH NUMBER=1 AND BODY_DESTINATION=0
      FROM INACTIVE_ROLLER_TABLES AND AT END OF ACTIVITY PUT BACK
      IN INACTIVE_ROLLER_TABLES.
  ATTRIBUTE MODIFICATIONS:
    BODY_DESTINATION OF ROLLER_TABLE := NUMBER OF PASS.
ACTIVITY:
    BODY_CONVEYANCE.
 ENDING_CONDITIONS:
    END_OF_BODY_CONVEYANCE=1.
 ENTITY FLOWS ( 1 OF THE FOLLOWING FLOWS):
    GET A  ROLLER_TABLE  WITH  BODY_DESTINATION >=
      NUMBER OF  ROLLER_TABLE FROM INACTIVE_ROLLER_TABLES AND
      AT END OF ACTIVITY PUT  BACK IN INACTIVE_ROLLER_TABLES.
    GET A ROLLER_TABLE WITH BODY_DESTINATION < 0
      FROM INACTIVE_ROLLER_TABLES AND AT END OF ACTIVITY PUT BACK
      IN INACTIVE_ROLLER_TABLES.
  ENTITY FLOWS:
    GET A ROLLER_TABLE WITH NUMBER = NUMBER OF
      ROLLER_TABLE (#1) + 1 AND BODY_DESTINATION = 0
      FROM INACTIVE_ROLLER_TABLES AND AT END OF ACTIVITY
      PUT  BACK IN INACTIVE_ROLLER_TABLES.
  ATTRIBUTE MODIFICATIONS:
    START_OF_BODY_CONVEYANCE := 1.
    BODY_DESTINATION OF ROLLER TABLE (#2) :=
      BODY_DESTINATION OF ROLLER TABLE (#1).
    BODY_DESTINATION OF ROLLER TABLE (#1) := 0.
ACTIVITY:
    LOAD_FROM_ROLLER_TABLE.
  ENDING_CONDITIONS:
    END_OF_LOAD_FROM_ROLLER_TABLE=1.
  ENTITY FLOWS (SELECTING BODY_DESTINATION OF ROLLER_TABLE IN
      INACTIVE_ROLLER_TABLES = NUMBER OF WORKSTATION IN
      WORKSTATION_POOL) :
    GET A  ROLLER_TABLE FROM INACTIVE_ROLLER_TABLES AND
      AT END OF ACTIVITY PUT  BACK IN INACTIVE_ROLLER_TABLES.
    GET A WORKSTATION WITH STATE_OF_LOADING_TABLE = 0 AND
      STATE_OF_LOADING_TABLE + STATE_OF_PROCESSING_TABLE +
      STATE_OF_UNLOADING_TABLE <= 1 FROM WORKSTATION_POOL AND AT
      END OF ACTIVITY PUT BACK IN WORKSTATION_POOL.
  ATTRIBUTE MODIFICATIONS:
```

```
  START_OF_LOAD_FROM_ROLLER_TABLE := 1.
  STATE_OF_LOADING_TABLE OF WORKSTATION := 1.
  BODY_DESTINATION OF ROLLER_TABLE := 0.
ACTIVITY:
  LOAD_FROM_LOADING_TABLE.
 ENDING_CONDITIONS:
  END_OF_LOAD_FROM_LOADING_TABLE=1.
 ENTITY FLOWS:
  GET A WORKSTATION WITH STATE_OF_PROCESSING_TABLE = 0
   AND STATE_OF_LOADING_TABLE = 1 FROM WORKSTATION_POOL
   AND AT END OF ACTIVITY PUT BACK IN WORKSTATION_POOL.
 ATTRIBUTE MODIFICATIONS:
   START_OF_LOAD_FROM_LOADING_TABLE := 1.
   STATE_OF_LOADING_TABLE OF WORKSTATION := 0.
   STATE_OF_PROCESSING_TABLE OF WORKSTATION := 1.
ACTIVITY:
  UNLOAD_FROM_PROCESSING_TABLE.
 ENDING_CONDITIONS:
  END_OF_UNLOAD_FROM_PROCESSING_TABLE=1.
 ENTITY FLOWS:
  GET A WORKSTATION WITH STATE_OF_PROCESSING_TABLE = 1
   AND STATE_OF_UNLOADING_TABLE = 0 FROM WORKSTATION_POOL
   AND AT END OF ACTIVITY PUT BACK IN WORKSTATION_POOL.
 ATTRIBUTE MODIFICATIONS:
  START_OF_UNLOAD_FROM_PROCESSING_TABLE := 1.
  STATE_OF_PROCESSING_TABLE OF WORKSTATION := 0.
  STATE_OF_UNLOADING_TABLE OF WORKSTATION := 1.
ACTIVITY:
  UNLOAD_FROM_UNLOADING_TABLE.
 ENDING_CONDITIONS:
  END_OF_UNLOAD_FROM_UNLOADING_TABLE=1.
 ENTITY FLOWS:
  GET A WORKSTATION WITH STATE_OF_UNLOADING_TABLE = 1
   FROM WORKSTATION_POOL AND AT END OF ACTIVITY
   PUT BACK IN WORKSTATION_POOL.
  GET A ROLLER_TABLE WITH NUMBER = NUMBER OF
   WORKSTATION AND BODY_DESTINATION = 0
   FROM INACTIVE_ROLLER_TABLES AND AT END OF ACTIVITY
   PUT  BACK IN INACTIVE_ROLLER_TABLES.
 ATTRIBUTE MODIFICATIONS:
   START_OF_UNLOAD_FROM_UNLOADING_TABLE := 1.
   STATE_OF_UNLOADING_TABLE OF WORKSTATION := 0.
   BODY_DESTINATION OF ROLLER_TABLE := - NUMBER OF WORKSTATION.
ACTIVITY:
 BODY_LEAVING.
 ENDING_CONDITIONS:
   END_OF_BODY_LEAVING=1.
ENTITY FLOWS:
   GET A ROLLER_TABLE WITH NUMBER=6 AND BODY_DESTINATION < 0
     FROM INACTIVE_ROLLER_TABLES AND AT END OF ACTIVITY PUT BACK
     IN INACTIVE_ROLLER_TABLES.
   AT END OF ACTIVITY PUT A PASS IN AVAILABLE_PASSES.
ATTRIBUTE MODIFICATIONS:
   END_OF_BODY_LEAVING :=1.
   NUMBER OF PASS := - BODY_DESTINATION OF ROLLER_TABLE.
   BODY_DESTINATION OF ROLLER_TABLE := 0.
END OF MODEL.
```

Figs. 5 to 8 depict the synthesis process of a CPN corresponding to the above model.
The result of step 1 is shown in fig. 5. The place set includes only one place for the passes because its only attribute is of constant value.

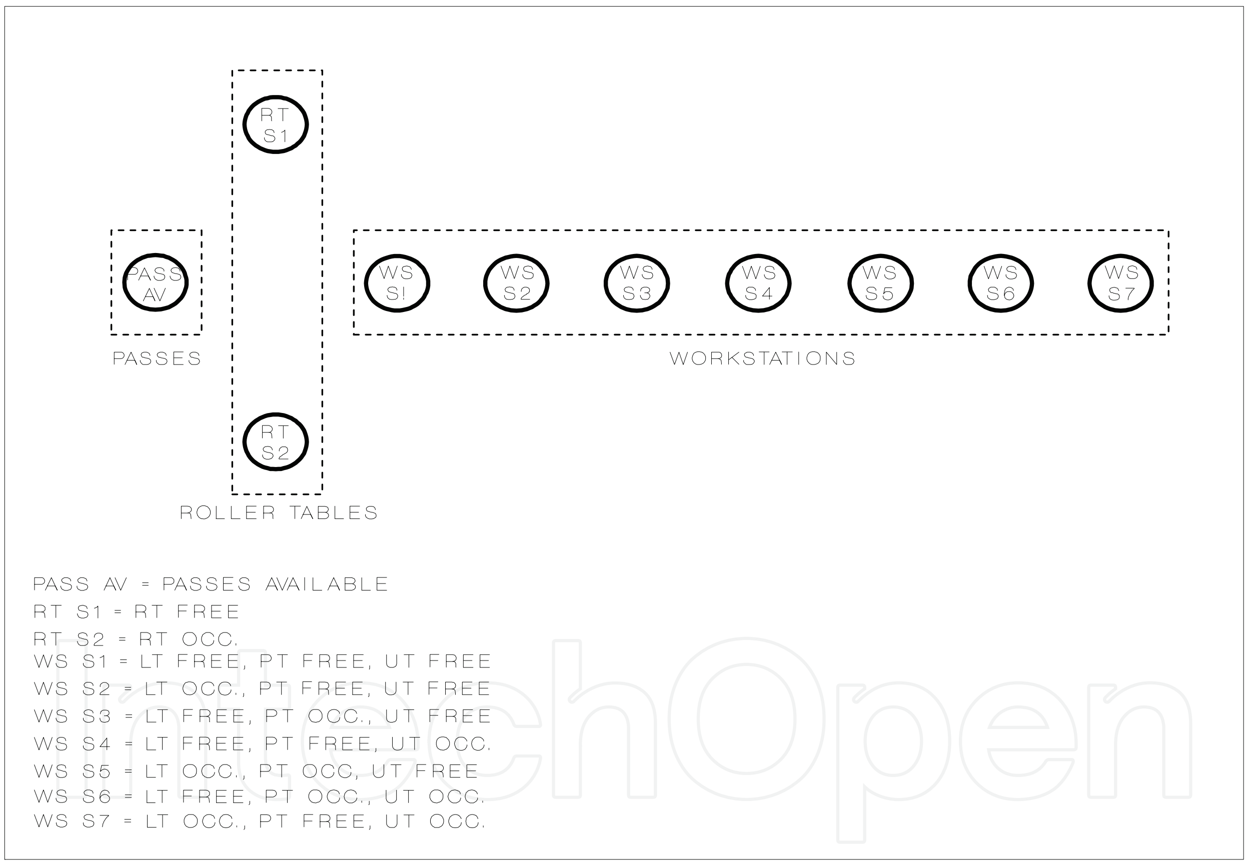
RT S1
PASS AV
PASSES
WS S!
WS S2
WS S3
WS S4
WS S5
WS S6
WS S7
WORKSTATIONS
RT S2
ROLLER TABLES
PASS AV = PASSES AVAILABLE
RT S1 = RT FREE
RT S2 = RT OCC.
WS S1 = LT FREE, PT FREE, UT FREE
WS S2 = LT OCC., PT FREE, UT FREE
WS S3 = LT FREE, PT OCC., UT FREE
WS S4 = LT FREE, PT FREE, UT OCC.
WS S5 = LT OCC., PT OCC, UT FREE
WS S6 = LT FREE, PT OCC., UT OCC.
WS S7 = LT OCC., PT FREE, UT OCC.

Fig. 5. Place set generated

Only two places were included for the roller tables because, in spite of the fact that the attribute corresponding to body destination can take 13 values, the roller table can only be in two states: free (RT S1) or occupied (RT S2). Workstations have seven different states (WS S1 to WS S7).
Fig. 6 depicts the transition sets corresponding to each one of the activities (step 2). There is one transition for each valid configuration of the system that allows the start of an activity. For example, activity LFRT is represented by the transitions LFRT1, LFRT2 and LFRT3.

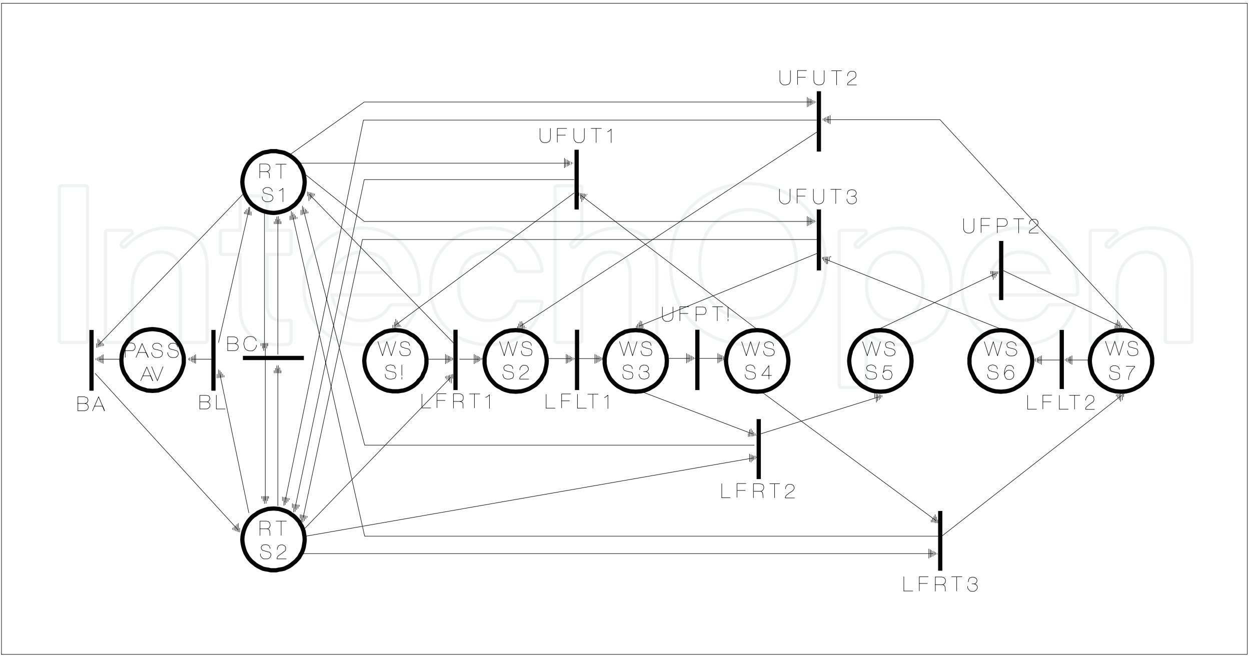

Fig. 6. Transition set generated by all activities

Fig. 7 summarizes the place and the transition colour sets (steps 3 and 4). First 10 colour sets correspond to places. All the places except one, the one corresponding to a roller table in an occupied state, have simple colours assigned because the values of the entity-variable attributes are uniquely determined by the states that the places represent. All the transitions have simple colours except transition BC that can fire in 36 different ways.

```
C(PASS AV)={<i>} i=1,…,6            C(BC)={<I,j>} i=1,…,6; j=1,…,6
C(RT S1)={<i>} i=1,…,6              C(LFRT1)={<i>} i=1,…,6
C(RT S2)={<i,j>} i=1,…,6; j=1,…,6   C(LFRT2)={<i>} i=1,…,6
C(WS S1)={<i>} i=1,…,6              C(LFRT3)={<i>} i=1,…,6
C(WS S2)={<i>} i=1,…,6              C(LFLT1)={<i>} i=1,…,6
C(WS S3)={<i>} i=1,…,6              C(LFLT2)={<i>} i=1,…,6
C(WS S4)={<i>} i=1,…,6              C(UFPT1)={<i>} i=1,…,6
C(WS S5)={<i>} i=1,…,6              C(UFPT2)={<i>} i=1,…,6
C(WS S6)={<i>} i=1,…,6              C(UFUT1)={<i>} i=1,…,6
C(WS S7)={<i>} i=1,…,6              C(UFUT2)={<i>} i=1,…,6
C(BA)={<i>} i=1,…,6                 C(UFUT3)={<i>} i=1,…,6
C(BL)={<i>} i=1,…,6
```

Fig. 7. Associated colours sets to places and transitions

Fig. 8 depicts the input and output functions (step 5). Because many of the colour sets of the transitions coincide with the colour sets of their input places, many of the functions are identity functions. Identity functions are represented by dashed arcs.

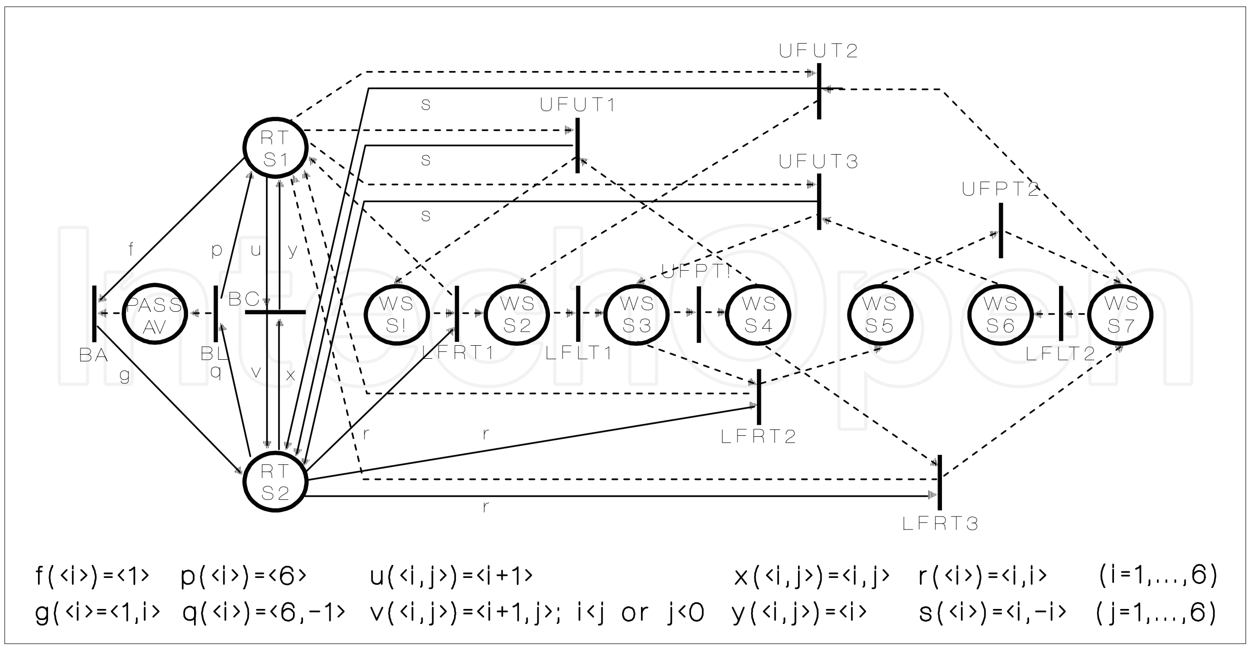
UFUT2
UFUT1
UFUT3
UFPT2
UFPT1
RT S1
RT S2
PASS AV
BA
BL
BC
WS SI
WS S2
WS S3
WS S4
WS S5
WS S6
WS S7
LFRT1
LFLT1
LFLT2
LFRT2
LFRT3
f
g
p
q
u
v
y
x
s
r
f(<i>)=<1> p(<i>)=<6> u(<i,j>)=<i+1> x(<i,j>)=<i,j> r(<i>)=<i,i> (i=1,...,6)
g(<i>=<1,i> q(<i>)=<6,-1> v(<i,j>)=<i+1,j>; i<j or j<0 y(<i,j>)=<i> s(<i>)=<i,-i> (j=1,...,6)

Fig. 8. Input-output functions generated

Function f takes from place RT SI the token that corresponds to roller table 1. Function g puts in place RT S2 a token that corresponds to roller table 1 occupied with a car body.
The body destination was taken by transition BA from place PASS AV (passes available). Functions *x* and u take from places RT S2 and RT SI, respectively, tokens corresponding to two consecutive roller tables, the first one occupied and the second one free. Functions y and v put tokens corresponding to those roller tables, but with the first one free and the second occupied, in places RT SI and RT S2, respectively. Function r, used in transitions corresponding to activity loading from roller table, takes from place RT S2 a token corresponding to a roller table whose number and body destination coincide. Function s, used in transitions corresponding to activity unloading from roller table, puts in place RT S2 a token corresponding to a roller table with a car body just processed, that is, the number of the roller table is the opposite of its body destination. In this example complex situations are not present in the input model, and therefore refinements (step 6) were not made to the net of fig. 8. This model is very close to that presented in (Colom et al., 1990), from where the real-life example is taken, but the functions of this model are simpler than those obtained by Colom.

7. Conclusion

A synthesis method for CPN was presented, a top-down technique in which the starting specifications of tasks are high-level expressed and predefined CPN modules are utilized. The method differs significantly from other methods published in that it preserves the formalism of CPN without imposing restrictions on the system modelled, and therefore it allows the modelling of shared resources and complex ordering and selecting criteria. The analysis of situations that can occur during the synthesis process is conducted using the philosophy of the activity-based approach for discrete-event systems. Input models of the synthesis process are specified in a natural-like language interface, which greatly facilitates

the expression of situations involving complex manipulations of items in waiting lines. These features also establish an advantage over other published methods.
The predefined modules, which deal with complex situations, are live and bounded; nevertheless the synthesized CPN model must be analyzed with existing mathematical methods for their validation and proof of properties; then it can be used for real-time control and simulation.

8. References

Aized, T., Takahashi, K. & Hagiwara I. (2007). Advanced multiple product flexible manufacturing system modelling using Coloured Petri Net. *Journal of Advanced Computational Intelligence and Intelligent Informatics.* Vol. 11, No. 6, pp. 715-723, ISSN: 1343-0130.

Arjona, E. & Bueno, G. (2007). Using simulation to integrate ordering and complex selecting criteria into Coloured Petri Net Models. *Agrociencia* Vol. 41, No. 8, pp. 883-901, ISSN: 1405-3195.

Badouel, E. & Darondeau, P. (2004). The synthesis of Petri nets from path-automatic specifications. *Information and Computation* Vol. 193, pp. 117-135, ISSN: 0890-5401.

Baldasari, M. & Bruno, G. (1991). PROTOB: an object oriented methodology for developing discrete event dynamic systems. *Computer Languages*, Vol. 16, No. 1, pp. 39-63, ISSN: 0096-0551.

Colom, J.M., Esparza, J., Martinez, J. & Silva, M. (1990). *DEMON: Design methods based on nets,* Esprit Basic Research Action 3148, University of Zaragoza, Spain, June 1990.

Da Silva, A., Montgomery, E. & Lima E. (2008). Flexible manufacturing systems modelling using high level Petri Nets. *Proceedings of ABCM Symposium Series in Mechatronics*, Vol. 3, pp. 405-413.

Der Jeng, M. & DiCesare, F. (1990). A review *of* synthesis techniques for Petri nets. *Proc. of the IEEE 2nd. International Conference on Computer Integrated Manufacturing,* pp. 348-355, Troy, NY, May 1990.

Diaz, M. (2009). Petri Nets: Fundamental Models, Verification and Applications, Wiley-ISTE, ISBN: 1848210795.

Ezpeleta, J. (1993). Análisis y síntesis de modelos libres de bloqueos para sistemas concurrentes, doctoral diss., University of Zaragoza, Spain.

He, D.W., Strege, B., Tolle, H. & Kusiak, A. (2000). Decomposition in automatic generation of Petri Nets for manufacturing system control and scheduling. *International Journal of Production Research,* Vol. 38, No. 6, pp. 1437-1457, ISSN: 0020-7543.

Jensen, K. (1981). Coloured Petri nets and the invariant method. *Theoretical Computer Science, Vol. 14*, pp. 317-336, ISSN: 0304-3975.

Jensen, K. (1991). Coloured Petri nets: A high level language for system design and analysis, In: *Lecture Notes in Computer Science, Advances in Petri nets 1990*, G. Rosenberg, Ed., pp. 342-416, Springer-Verlag, ISBN: 978-3-540-53863-9, Berlin.

Khadka, B. (2007). Transformation of live sequence charts to Colored Petri Nets, *masters project report,* University of Massachusetts Dartmouth, USA.

Kreutzer, *W. (1986). System Simulation Programming Styles and Languages*, Addison-Wesley, ISBN: 0-201-12914-0, Reading, MA, USA.

MengChu, Z. (2009). *System modeling and control with resource-oriented Petri Nets,* CRC Press, ISBN: 978-1-4398-0884-9, Boca Raton, FL, USA.

Micovsky, A., Sesera, L., Veishab, *M. &* Albert, M. (1990). TORA: A Petri net based tool for rapid prototyping of FMS control systems. *Computers in Industry, Vol. 15, No. 4*, pp. 279-292, ISSN: 0166-3615.

Shang, D., Burns, F., Koelmans, A., Yakovlev, A. & Xia, F. (2004). Asynchronous system synthesis based on direct mapping using VHDL and Petri nets. *IEE Proceedings Computers and Digital Techniques,* Vol. 151, No. 3, ISSN: 1751-8601.

Yeung, D.S., Shiu, S.C.K. & Tsang, E.C.C. (1999). Modelling flexible manufacturing systems using weighted Fuzzy Coloured Petri Nets. *Journal of Intelligent and Fuzzy Systems,* Vol. 7, No. 2, pp. 137-149, ISSN: 1064-1246.

Zhi-Jun, D., Jun-Li, W & Chang-Jun, J. (2008). An approach for synthesis Petri nets for modeling and verifying composite web service, *Journal of Information Science and Engineering*, Vol. 24, pp. 1309-1328, ISSN: 1016-23.

Zhou, M., DiCesare, F., & Desrochers, A. (1992). A hybrid methodology for synthesis of Petri net models for manufacturing systems. *IEEE Trans. on Robotics and Automation,* Vol. 8, No. 3, pp. 350-361, ISSN: 1042-296X.

Appendix. Formal Grammar of the Language

Before giving the grammar of the language, we will explain the logic behind it, describe the different ways in which its elements can be used, and give some general examples of its use. These examples complement the language features showed in the automotive workshop example given in section 6.

The language was designed analyzing all possible situations that can occur when modeling a real life system using the activity-approach paradigm. As was said before, besides of declarations, the language uses only two types of statements. The first one is used for entity (material and resource) flows and the second one for attribute (state) modifications. Interactive actions are modeled by explicit starting and ending conditions in the activities and these conditions can use both internal and external variables. Only one type of statement is used for entity flows because the number of different ways in which is possible to store, or retrieve, an entity in a waiting line (storage areas, buffers, and queues) is very small. Storing and retrieving depend heavily on the ordering criteria of the waiting lines involved and, excluding nonsense ordering criteria, a waiting line can be ordered in only a few ways. Only one type of statement is used for attribute modifications because the only modifications that attributes are susceptible to are assignments. Flows of entities from waiting lines to activities can be individual or multiple. Individual flows can be unconditional or conditioned to attribute values of entities or to entity positions in waiting lines when these have associated ordering criteria (overriding). Conditions on attribute values can be absolute, relative to attribute values of other entities that already are in the activity, or relative to values of the model variables. Multiple flows consist of a fixed or variable number of individual flows. Priorities can be given to the entities required for a particular activity (alternate flows). Flows of entities from activities to waiting lines can only be individual. Alternate storage waiting lines can be specified based on attribute values of the entities. Also, when the output waiting lines have associated ordering criteria, a position for the storage of an entity can be specified (overriding). Assignments can be conditional or unconditional. Values assigned are the result of the evaluation of expressions that may have as operands attribute names, constants, variable names, and intrinsic and extrinsic functions. One assign statement will affect all similar entities that satisfy the conditions of the statement (implicit repeat). There is an intrinsic function (#) for the counting of entities included in an activity and that meet specific attribute characteristics. This function is of particular importance when using multiple flows with a variable (unknown) number of individual flows.

Specification of a model consists of six parts or sections. In the first section is given the name of the model. In the second section are given the names of the entities and attributes and the admissible values of the attributes. In the third section are given the names of the waiting lines, and their ordering criteria and admissible entities. In sections four and five, are given the names of the internal and external variables of the system. In the sixth section, the activities that make up the system are described. All declarations and statements must end with a point. Declarations are used in all sections, and statements are used only in the sixth section that corresponds to activity descriptions. All the names should start with an alphabetic character and may be followed by a string of alphanumeric or underscore characters. Entities may have or not attributes. Variables are defined in a similar way as entities. Internal variables correspond to endogenous variables and external variables correspond to exogenous or interactive variables. Variables for which admissible values are not specified are assumed boolean whose default admissible values are zero and one. Waiting lines can have an ordering criterion or a length specified. When it is not specified an ordering criterion the waiting line is ordered FIFO. The names of entities allowed in waiting lines can be pluralized to improve readability. Activities can have multiplicity. Activity multiplicity must be evaluated to an integer number that indicates the maximum number of concurrent occurrences of the activity; default multiplicity is one. Starting and ending conditions are boolean expressions that use relational expressions of variables and constants. Ending conditions and activity durations are mutually exclusive. Activity durations can be obtained from random variates using external variables or intrinsic functions. A flow statement may consist of one or more individual entity flows. Each individual flow can be a generator, a transmitter or a consumer of entities. The number of individual flows in a flow statement can be constant or variable. Priorities can be specified between individual flows as well as alternate choices. Entities in a flow can be conditioned by means of their attribute values to a constant or to other entities attribute values, in an absolute or a relative way. In addition, it is possible to override ordering criteria of the waiting lines from where the entities are removed, and it is possible to specify alternate destination waiting lines for the storage of entities. The scope of a flow statement is used to select a subset of a set of individual flows. The cardinality of the subset can be fixed or variable within a range. The selecting option is used to define local relations among entities. Repetition factors condense model specifications when identical flows are used. Many of the words included in the flow statements are optional. Finally, attribute modifications can only consist of a conditional or an unconditional assignment.

Examples of valid entity definitions are the following:

```
ENTITIES:
  CAR WITH THE FOLLOWING ATTRIBUTES:
    MAKE (ADMISSIBLE VALUES FORD OR CHRYSLER)
    PASSENGER_CAPACITY (ADMISSIBLE VALUES 2 TO 5).
  ELEVATOR.
  CRANE.
```

Examples of valid variable definitions are the following:

```
INTERNAL VARIABLES:
   PRODUCT_GENERATED.
EXTERNAL VARIABLES:
   USER_ANSWER (ADMISSIBLE VALUES HIGH OR MEDIUM OR LOW).
```

Examples of valid waiting line definitions are the following:

```
WAITING LINES:
   WAREHOUSE CONTAINS BOXES.
   ASSEMBLY_LINE (FIFO) CONTAINS CAR_BODIES AND ITS  MAXIMUN LENGHT IS
    10.
   WAITING_ROOM CONTAINS PATIENTS ON DESCENDING ORDER BY
     ILLNESS_CONDITION.
```

Examples of valid starting and ending conditions definitions of activities are:

```
ACTIVITY:
   BEST_QUALITY_CONTROL.
 STARTING CONDITIONS:
   USER_ANSWER=HIGH.
 ACTIVITY DURATION: 3.

ACTIVITY:
    DIRECT_ASSEMBLY.
 STARTING CONDITIONS:
    INFRARED_SENSOR=1.AND.RECOGNITION_TO_BE_DONE=0.
 ENDING CONDITIONS:
    END_OF_ASSEMBLY=1.
```

Examples of valid flow statements are the following:

```
ENTITY FLOWS:
 GET A WORKER WITH SKILL>2 FROM CREW.
 GET A BIT WITH SIZE=1/2 FROM THE BIT_PALETTE AND AT END OF ACTIVITY PUT
    BACK IN THE BIT_PALETTE.
 AT END OF ACTIVITY PUT A PASS IN AVAILABLE_PASSES.

ENTITY FLOWS  (FROM 30 TO 150 OF THE FOLLOWING FLOWS):
 (AT MOST 20 TIMES) GET A PASSENGER WITH CLASS=FIRST FROM
   PASSENGER_LIST AND AT END OF ACTIVITY PUT IN IMMIGRATION_LANE.
 (AT MOST 130 TIMES) GET A PASSENGER WITH CLASS=SECOND.OR.CLASS=THIRD
   FROM PASSENGER_LIST AND AT END OF ACTIVITY PUT IN IMMIGRATION_LANE.

ENTITY FLOWS  (SELECTING ASSEMBLY_STATE OF ASSEMBLY_SITE IN
                  ASSEMBLY_TABLE = TYPE_OF_PART OF PART IN STORAGE_TABLE  -1):
   GET A PART FROM STORAGE_TABLE.
   GET AN ASSEMBLY_SITE FROM ASSEMBLY_TABLE AND AT END OF ACTIVITY
     PUT BACK IN ASSEMBLY_TABLE.
```

Examples of valid attribute modifications are the following:

```
ATTRIBUTE MODIFICATIONS:
   TOTAL_NUMBER_OF_JOBS := TOTAL_NUMBER_OF_JOBS +1.
   JOBS_DONE OF WORKER := JOBS_DONE OF WORKER +1.
   COLOR OF CAR := COLOR OF PAINT_ORDER.
   IF ASSEMBLY_STATE OF ASSEMBLY_SITE = 4 THEN POSITION OF ROBOT1 := 3.
```

Now, we will give the definition of the language. A computer language is defined by its formal grammar. A formal grammar defines rules for building syntactically correct sentences in the language. Sentences are made up of strings of characters that are logically combined into grammar elements using grammar rules. Grammar rules are specified by means of productions that state how a grammar element can be composed from other grammar elements. Each production defines a grammar element and has two sides separated by the symbol "::=". The left hand side of the production is the grammar element defined (represented by a nonempty string of characters enclosed in triangular parenthesis) and the right hand side its definition. This definition consists of grammar elements, character strings (letter, digits, keywords, punctuation marks, operators, etc.), and auxiliary symbols with specific meanings (square brackets indicate optional items, three periods indicate that the preceding element can be repeated one or more times, and a vertical bar indicate a choice of items). For example, the productions:

```
<identifier> ::= <letter> [<letter> | <digit> | _ ] ...
<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
              U | V | W | X | Y | Z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<global criterion of ordering> ::= FIFO | LIFO
```

state that the grammar element called "identifier" is built up using a letter that may be followed by one or more letters or digits or the underscore character, and that the grammar element called "global criterion of ordering" consists of one of the keywords FIFO or LIFO. Note that the grammar elements called "letter" and "digit" used in the first production need to be defined because words or phrases used to represent grammar elements do not have any real meaning until defined.
Following, it is given the formal grammar of the language.

```
<model>::=MODEL NAME:<model name><period>
             <entities definition>
             [<internal variables definition>]
             [<external variables definition>]
             <waiting lines definition>
             <activity definition>...
             END OF MODEL <period>
<model name> ::= <identifier>
<identifier> ::= <letter> [<letter> | <digit> | _ ] ...
<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T |
             U | V | W | X | Y | Z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
<period> ::= .
<entities definition> ::= ENTITIES: <entity> <period> [<entity> <period>]...
<entity> ::= <entity name> [WITH THE FOLLOWING ATTRIBUTES: <attributes>]
<entity name> ::= <identifier>
<attributes> ::= <name and admissible values of the attribute>
                 [<name and admissible values of the attribute>]...
<name and admissible values of the attribute> ::= <attribute name> [(<admissible values>)]
<attribute name> ::= <identifier>
<admissible values> ::= [ADMISSIBLE VALUES] <range of values> |
                        [ADMISSIBLE VALUES] <specific values>
<range of values> ::= <integer> TO <integer>
<integer> ::= [+|-] <string of digits>
<string of digits> ::= <digit> [<digit>]...
<specific values> ::= <integer or literal> [OR <integer or literal>]...
<integer or literal> ::= <integer> | <literal>
<literal> ::= <identifier>
<internal variables definition> ::= INTERNAL VARIABLES: <internal variable><period>
                                    [<internal variable> <period>]...
<internal variable> ::= <internal variable name> [(<admissible values>)]
<internal variable name> ::= <identifier>
<external variables definition> ::= EXTERNAL VARIABLES: <external variable> <period>
                                    [<external variable> <period>]...
<external variable> ::= <external variable name> [(<admissible values>)]
<external variable name> ::= <identifier>
<waiting lines definition>::= WAITING LINES: <waiting line> <period> [<waiting line>
                              <period>]...
<waiting line> ::= <waiting line name>
                   [(<global criterion of ordering>)]
                   CONTAINS <admissible entity> [AND <admissible entity>]...
                   [AND ITS MAXIMUM LENGTH IS <integer>]
<waiting line name> ::= <identifier>
<global criterion of ordering> ::= FIFO | LIFO
<admissible entity> ::= <pluralized entity name> [<local criterion of ordering> BY
                        <attribute name>]
<pluralized entity name> ::= <entity name> [S | ES]
<local criterion of ordering> ::= ON ASCENDING ORDER | ON DESCENDING ORDER
<activity definition> ::= ACTIVITY: <activity name> <period>
                          [MULTIPLICITY: <expression of variables> <period>]
                          [STARTING CONDITIONS: <condition of variables> <period>]
                          [ENDING CONDITIONS: <condition of variables> <period>]
                          <flow statement> <period> [<flow statement> <period>]...
                          [<attribute modifications>]
                          [ACTIVITY DURATION: <expression of variables> <period>]
<activity name> ::= <identifier>
<condition of variables> ::= <boolean term of variables>
                             [.AND.<boolean term of variables>]
```

```
<boolean term of variables> ::= <boolean factor of variables>
                                      [.OR.<boolean factor of variables>]
<boolean factor of variables> ::= <relation of variables> | (<condition of variables>)
<relation of variables> ::= <expression of variables> <relational operator>
                               <expression of variables>
<expression of variables> ::= <term of variables>
                                   [+<term of variables> | -<term of variables>]
<term of  variables> ::= <factor of variables>
                            [*<factor of variables> | /<factor of variables>]
<factor of variables> ::= <atom of variables> | <factor of variables> @ <atom of variables>
<atom of variables> ::= <constant> |
                            <internal variable name> |
                            <external variable name> |
                            <function of variables> |
                            (<expression of variables>)
<relational operator> ::= > | < | = | <> | >= | <=
<constant> ::= <integer number> | [+|-].<string of digits> |
                 <integer number>.<string of digits> | <literal>
<function of variables> ::= <function name> (<expression of variables>)
<function name> ::= <identifier>
<flow statement> ::= ENTITY FLOWS
                        [(<scope and select condition>)]:
                        <flow specification>...
<scope and select condition> ::= <scope> |<scope> <select condition> | <select condition>
<scope> ::= [ONLY] <expression> OF THE FOLLOWING FLOWS |
              FROM <expression> TO <expression> OF THE FOLLOWING FLOWS
<select condition> ::= SELECTING <relation of entities>
                          [AND <relation of entities>]...
<relation of entities> ::= <expression of entities> <relational operator>
                              <expression of entities>
<expression of entities> ::= <term of entities> [+<term of entities> |-<term of entities>]
<term of entities> ::= <factor of entities> [*<factor of entities> | /<factor of entities>]
<factor of entities> ::= <atom of entities> | <factor of entities> @ <atom of entities>
<atom of entities> ::= <constant> |  <attribute name> OF <entity name>
                          IN <waiting line name>
<flow specification> ::= [<repetition factor>] [IF <condition>] <flow>
<repetition factor> ::= ([AT MOST] <expression> TIMES)
<condition> ::= <boolean term> [.AND. <boolean term>]
<boolean term> ::= <boolean factor> [.OR. <boolean factor>]
<boolean factor> ::= <relation> | (<condition>)
<relation> ::= <expression> <relational operator> <expression>
<expression> ::= <term> [+<term> | -<term>]
<term> ::= <factor> [*<factor> | /<factor>]
<factor> ::= <atom> | <factor>  @ <atom>
<atom> ::= <constant> |
             <internal variable name> |
                  <external variable name> |
                  <number of entities> |
                   <entity attribute> |
                   <function> |
                   (<expression>)
<number of entities> ::= # <pluralized entity name> [WITH <entity condition>]
<entity condition> ::= <expression of attributes> <relational operator> <expression> [AND
                       <expression of attributes> <relational operator> <expression>]... |
                       (<entity condition>)
<expression of attributes> ::= <term of attributes> [+<term of attributes> |
                               -<term of attributes>]
<term of attributes> ::= <factor of attributes>
                         [*<factor of attributes> | /<factor of attributes>]
<factor of attributes> ::= <atom of attributes> | <factor of attributes> @ <atom of attributes>
<atom of attributes> ::= <constant> |  <attribute name>
<entity attribute> ::= <attribute name> OF <entity name> [(# <string of digits>)]
<function> ::= <function name> (<expression>)
<flow> ::= <consumption flow> | <transmission flow> | <generation flow>
<consumption flow> ::= <input flow> <period>
<input flow> ::= GET [A | AN] <entity name>[WITH <entity condition>]
                 FROM [THE] [<position> OF THE] <name of waiting line>
<position> ::= FRONT | END
<transmission flow> ::= <input flow> AND [AT END OF <identifier>]
                        PUT [BACK] <output flow> [,<output flow>]... <period>
<output flow> ::= IN [<position> OF ] <name of waiting line> [IF <condition>]
<generation flow> ::= [AT END OF <identifier>] PUT [A | AN]
                      <entity name> <output flow>
<attribute modification> ::= ATTRIBUTE MODIFICATIONS:
                             <modification statement> <period>
                             [<modification statement> <period>]...
<modification statement> ::= [IF <condition> THEN]
                              <left side of assignment> := <expression>
<left side of assignment> ::= <internal variable name> | <external variable name> |
                              <entity attribute>
```

End of the formal grammar definition.