SIMULATION OF PETRI NETS WITH MULTIPLE INSTANCES USING DEVS

Sergio GIRO*

FaMAF – CONICET
Ciudad Universitaria
Córdoba
sgiro@famaf.unc.edu.ar

Claudia FRYDMAN

LSIS - UMR CNRS 6168 – Université Aix-Marseille III
Avenue Escadrille Normandie-Niemen
13397 Marseille CEDEX 20
claudia.frydman@lsis.org

ABSTRACT : *Petri nets with multiple instances (PNMIs) were proposed as a low-level formalism being suitable to define semantics for high-level workflow languages. So, verification techniques for PNMIs can be used for any high-level workflow formalism whose semantics can be defined using PNMIs. In this paper, we show how PNMIs can be extended to allow timing constraints. We also show how timed PNMIs can be encoded into DEVS models, thus making an important contribution towards the simulation of workflow systems using PNMIs.*

KEYWORDS : *simulation, DEVS, Petri nets with multiple instances, workflows*

1. INTRODUCTION

Workflow models experienced an enormous evolvement in later years, given the widespread use of the Service Oriented Architecture (SOA), and the fact that the *orchestration* of services is usually specified using workflows.

DEVS models (Zeigler *et al.*, 2000) are widely used in both academia and industry, and a lot of research related to DEVS has been carried out since the formalism was developed in the early 70's. So, the simulation of workflow models using DEVS is a highly desirable technique for the development of software systems.

Petri nets with multiple instances, or PNMIs (Giro and Frydman, 2006) were proposed as a low-level formalism being suitable to define semantics for high-level formalisms. So, the techniques available for PNMIs can be used for any high-level formalism whose semantics could be put in terms of PNMIs. Briefly speaking, PNMIs are sets of Petri nets that are able to be synchronized according to certain rules. So, our models have more expressive power with respect to the multiple instances as defined in the YAWL language (see (van der Aalst and ter Hofstede, 2005)).

The development of a new low-level formalism was motivated by the fact that many high-level formalisms exist, often covering disimilar features (Giro, 2007). Moreover, the level of these languages is very high, thus making it difficult to develop fundamental theories. In addition, as explained in (Giro, 2007), a low-level formalism can be extended to deal with other aspects (for example, finite-state automata can be extended to timed automata) without changing the underlying formalism.

* Part of this research was carried out at LSIS. Partially supported by STMicroelectronics, Z.I. 13106 ROUSSET, France.

Initially, PNMIs were conceived as a modelling language intended for formal verification and simulation. In this paper we address the simulation of PNMI models. The existence of simulation techniques for PNMIs implies the existence of simulation techniques for every high-level workflow language whose semantics can be defined using PNMIs, since any behaviour in the high-level language has a corresponding behaviour (given by the semantics) in a PNMI and, conversely, each behaviour in the underlying PNMI has a conceptual meaning. A methodology for the evaluation of systems using PNMIs is sketched in figure 1.

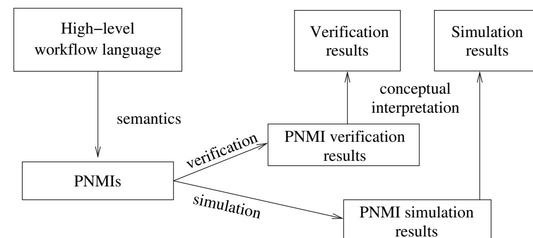

Figure 1. Evaluation of workflow systems using PNMIs

We believe that simulation is an excellent complement of formal verification, since simulation is useful to evaluate the *behaviour* of the system while formal verification is useful to prove the *correctness* of the system with respect to properties. When looking for formalisms for workflow simulation, we found that many tools are available for DEVS models (see, for instance, (Zacharewicz, 2006)). In addition, hierarchical DEVS models are close to PNMIs, since in such models several instances of an atomic DEVS model can be simulated in parallel.

Since many systems in different domains require timing constraints in order to be properly modelled, and such constraints can be expressed in DEVS models, we start extending PNMIs to allow the specification of timing con-

straints. We introduce such constraints in the underlying Petri nets and show how they can be used to model the time required by workflow tasks. In addition, an illustrative example of the use of timed PNMIs is presented in order to show the suitability of the formalism. After defining timed PNMIs, we show how they can be encoded into DEVS models, using the example in order to make the explanation easier.

It is worth noting that the PNMIs as presented here differ from the ones presented in (Giro and Frydman, 2006), but the fundamental concept in both approaches is the same (a set of Petri nets related by a hierarchy, in which the hierarchy restricts subsets of instances being able to execute a synchronization) and the suitability to model workflow systems explained in (Giro and Frydman, 2006) can be easily verified to hold for the PNMIs as presented here. In the conclusion we compare both approaches.

While in (Cristiá, 2007) DEVS models are encoded into a formal specification language (TLA+) our encoding goes the other way round. In order to encode PNMI models into DEVS, we must consider the different aspects considered by both formalisms. In addition to the timing constraints, nondeterminism is a key difference between both formalisms. While PNMIs are able to model behaviours in which an event can lead to different states, DEVS models specify internal and external transition *functions* in such a way that by fixing the external event, the amount of elapsed time and the actual state, the next state is *univocally* determined. In order to cope with this difference, either PNMIs should be restricted to behave deterministically or an extension to DEVS including nondeterminism must be considered. In this paper, we explore the former option. The PNMIs restricted to deterministic behaviours are still useful: after all, DEVS can only model deterministic behaviours and they were used to model a large variety of systems. However, encoding of nondeterministic behaviours from PNMIs into DEVS models is an interesting work to be developed in the future.

The organisation of the paper is as follows: in section 2, we present the basic concepts about DEVS and PNMIs. In section 3, we extend PNMIs to allow timing constraints. In section 4, we show how the extended PNMIs introduced in section 3 can be used to model workflow systems. In section 5, we present our encoding from PNMIs to DEVS and prove an interesting property related to the consistence of the generated DEVS models. Finally, the conclusion summarizes the content of the paper and proposes further work.

2. PETRI NETS WITH MULTIPLE INSTANCES AND DEVS

In this section we introduce PNMIs and recall the basic aspects of DEVS.

2.1. Petri nets with multiple instances

We start with an informal explanation of the PNMIs based on three toy examples.

2.1.1. Informal explanation of PNMIs

A PNMI model is obtained by composing several Petri net models. Many instances of these Petri net models run concurrently, these instances being related by a hierarchy.

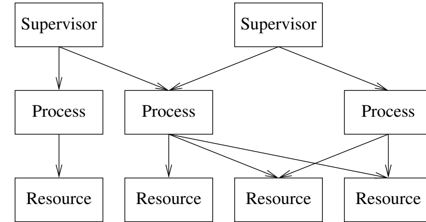

Figure 2. Example of a PNMI hierarchy

As an example, figure 2 shows the hierarchy relating nine instances whose Petri net models (not shown in the figure) are named Supervisor, Process and Resource. The instances are represented by boxes in the figure. An arrow connecting two instances indicates that either a process is controlled by a supervisor or a resource may be used by a process during its execution. The transitions in the Petri net models we consider are labelled. Labels are called *actions*. We say that a Petri net executes an action a if it executes a transition labelled with a. The PNMI changes its state according to *compound transitions*, which are specified separately. In the execution of a compound transition, many instances execute an action simultaneously. Each compound transition indicates how the instances must be related by the hierarchy (an example of such restrictions being "the process and the supervisor must be such that the process is controlled by the supervisor"). So, the hierarchy determines whether a set of instances is able to perform a compound transition or not. Three examples of compound transitions are shown in figure 3.

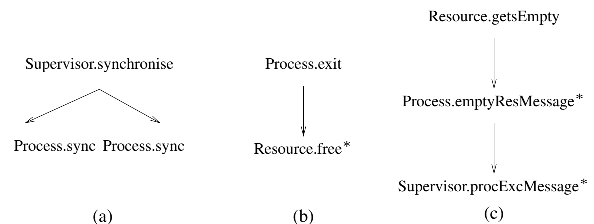

Figure 3. Examples of compound transitions

The compound transition (a) can be executed only if a supervisor S can execute the action $synchronize$ (that is, if S can execute a transition labelled with $synchronize$) and two *different* process P and Q supervised by S (as indicated by the hierarchy) can execute the action *sync*. The resulting state of the PNMI is obtained by taking the states of S, P and Q after the execution of the corresponding transitions while leaving the states of the other instances unchanged. The compound transition (b) can be executed

only if a process P can execute the action *exit* and *all* the resources R_i that may be used by P can execute the action *free*. The asterisk in *free* indicates that all instances complying with the hierarchy must be considered. Again, the next state is obtained by executing the corresponding transition in every involved instance. The compound transition (c) can be executed only if a resource is able to execute the action $getsEmpty$. This compound transition indicates that, when a resource gets empty, all the process that may need it are notified, as well as their supervisors. This example illustrates that the arrows in the specification of the compound transition may not have the same direction as the arrows in the hierarchy: barely speaking, the arrows in the specification indicate the order in which the instances are chosen in order to execute a transition. First of all, a resource is chosen. Then, we pick all the processes related to this resource. Finally, we pick all the supervisors related to these processes.

2.1.2. Formal definition of PNMIs

Now that the reader has an intuitive understanding of how PNMIs works, we introduce the formal definition. First of all, we recall the definition of traditional Petri nets and the firing rule.

Definition 1 (Petri net)**.** A Petri net is a 5-uple (P, T, A, w, M_i) where P is a set of places, T is a set of transitions, $A \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, $w : A \rightarrow \mathbb{N}$ assigns a weight to each arc, and the initial marking M_i is a mapping $M_i : P \rightarrow \mathbb{N}$.

Definition 2 (Marking)**.** A marking M for a Petri net is a mapping $M : P \rightarrow \mathbb{N}$, where P is the set of places of the Petri net.

A Petri net with labelled transitions is a pair (N, l) where $l : T \rightarrow \mathcal{L}$ is a function from the transitions of the Petri net N to a set $\mathcal{L}$ of labels.

Given a transition t of a Petri net with a set of arcs A, the sets of input places (output places, resp.) of t is the set $\bullet\, t$ ($t\, \bullet$, resp.) of places s such that $(s, t) \in A$ ($(t, s) \in A$, resp.).

Definition 3 (Firing rule for Petri nets)**.** We say that a marking M leads to a marking M' firing transition t if there are at least $w((p, t))$ tokens in every input place p of t, and if $M'(p) = M(p) - w'((p, t)) + w'((t, p))$ for every place p, where $w'((a, b)) = w((a, b))$ if $(a, b) \in A$ and $w'((a, b)) = 0$ if $(a, b) \notin A$.

Since DEVS do not allow nondeterministic behaviours, we restrict the PNMIs to behave deterministically. A Petri net is *deterministic on actions* if for every action a and markings M, M', M'' such that M leads to M' by firing a transition t labelled with a and M leads to M'' by firing a transition t' labelled with a, then $M' = M''$. We assume that all the Petri nets are *deterministic on actions*. The determinism on actions can be verified, for instance, by constructing a covering tree for the Petri net. So, we suppose that the determinism was verified beforehand. Note that, for the purpose of simulation, the nondeterminism can be avoided by properly relabelling the actions according to the conceptual meaning of the different nondeterministic options. This is particularly convenient in the case of fault-tolerant systems, where nondeterministic behaviours are often related to errors and exceptions. As an example, if the action $Process.exit$ can lead either to a state in which the process effectively finishes or to a state in which the termination has not been successful because of an error, the nondeterminism can be avoided using two labels $successfulExit$ and $badExit$, and two compound transitions, in order to make explicit the fact that, at a given point in the execution of the process, there are two possible courses of execution.

In the following, we will use some standard definitions related to graphs. We describe the definitions we use in a very brief manner since these terms should be easily understood by the reader. If it is not the case, see (Valiente, 2002). A directed acyclic graph (DAG) is a directed graph having no directed cycles (undirected cycles are allowed). An arc a in a directed graph connects two nodes n_1 and n_2 iff either $a = (n_1, n_2)$ or $a = (n_2, n_1)$. For every DAG D, we define the following terms: a root of D is a node with no incoming arcs. The height of D is the length of the largest path in D. A node n is at depth d if the largest path from any root to n is d. Note that two nodes at the same depth cannot be connected by an arc.

Having settled the terminology for graphs, we start to define PNMIs. As we have seen in the introduction, PNMIs have compound transitions that allow the different Petri nets to synchronize. A compound transition is a directed acyclic graph whose nodes are labelled. These labels are of the form $PetriNetName.action\,[^*]$ (the asterisk is not always present).

Definition 4 (PNMI)**.** A Petri net with multiple instances is a set $\mathcal{N}$ of Petri net names, a function m mapping elements in $\mathcal{N}$ to Petri nets with labelled transitions, a directed acyclic graph $\mathcal{H}$ whose nodes are labelled with elements from N specifying the hierarchy, and a set $\mathcal{T}$ of compound transitions.

The function m is used to obtain the definition of a Petri net (its places, transitions, arcs, etc.) given its name.

A marking for a PNMI is a directed acyclic graph whose nodes are labelled with pairs (n, M), where n is a Petri net name and M is a marking for the corresponding Petri net $m(n)$. The labelling must be such that by ignoring the markings M, the directed acyclic graph obtained is the hierarchy $\mathcal{H}$.

In order to define the firing rule for PNMIs, we must define the sets of Petri nets that are able to execute a compound transition. There are some particular details that

complicate the definition. Consider the compound transition in figure 4 (a).

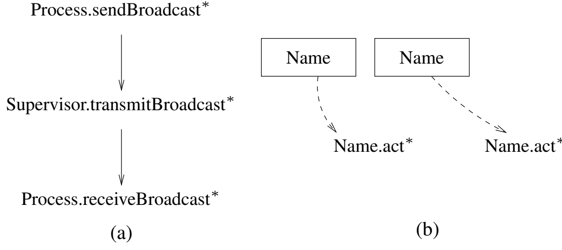

Figure 4. Subtleties concerning PNMIs

In this transition, a process sends a broadcast message to every process sharing a supervisor. In the asterisk at the bottom of the transition, we do not want to include the process sending the message, because if this process were included, it should execute both the actions $receive$ and $send$. In these cases, when choosing all the instances corresponding to an asterisk we will say that the instance is not compromised. We will also say that an instance is not compromised if the hierarchy does not relate such an instance to the instances considered before (recall that the choices are done in a "top-bottom" fashion). In addition, in order to calculate the marking after the firing, we must keep the information related to the roles played by each instance while firing the compound transition. For example, suppose that there are two labels $name.act1$ and $name.act2$ in the same compound transition. In this case, we should keep information regarding what instance of the Petri net $name$ executes the action $act1$ and what instances executes $act2$. So, we will say that the subgraph S is enabled to fire a compound transition T under role R, R being a function $R : S \to N$, where N are the nodes of the compound transition.

The following definitions are rather technical, and we give them for the sake of completeness. The reader not interested in the formal definition of PNMIs can safely skip the rest of this subsection if the intuitive explanations about the firing rule for PNMIs suffice for its purposes.

All the sets of all subgraphs of a marking M for which a compound transition T is enabled taking into account that the set Q of nodes is not compromised can be inductively defined as follows.

If the compound transition T is a directed acyclic graph of height 0, then T is simply a set of nodes with labels of the form $PetriNetName.action\,[\,^*\,]$. T is enabled in S if no elements in S are in Q and there is a function R mapping the elements in S to the nodes in T such that: **(1)** for every $s \in S$, if $R(s) = n$ then the label of s in M is a pair $(name, M_s)$ such the label $l(n)$ is of the form $name.act\,[\,^*\,]$ and a is enabled in M_s, **(2)** for every $s \in S$, if $R(s) = n$ and $R(s') = n$ then either $s = s'$ or $l(n)$ is of the form $name.act^*$ (this condition ensures that different instances with the same name as in in the compound transition in figure 2 (a) map to different instances in the marking) and **(3)** if there is an element in T of the form $name.act^*$ then every node with a label of the form $(name, M)$ is either in S or in Q (this condition ensures that labels with asterisk apply to all instances, excepting for the ones not being compromised. Note that in the cases in which there more than one label with asterisk for the same Petri net name —as shown in figure 3 (b),— each compromised instance can be mapped to different asterisk labels).

For the inductive part of the definition, consider a compound transition T being a directed acyclic graph of height $H > 0$ and let S be a subgraph of height $H > 0$. Let T' be the directed acyclic graph obtained by removing the nodes in T at depth H, T'' be the graph obtained by removing the nodes in T at depth less than H and similarly we define S' and S'' for S. Let U be the set of nodes n such that there is no undirected path from S' to n in $\mathcal{H}$. Then, T is enabled in S if **(1)** T' is enabled in S' under some role function R', **(2)** T'' is enabled in S'' under some role function R'' having the set $Q \cup U \cup T'$ as the set of states not being compromised and **(3)** $R'(s) = n$ and $R''(t) = m$ implies that, if there is an arc from n to m, then there exists an arc connecting s and t in $\mathcal{H}$. The resulting role function is: $R(s) = R'(s)$ if $s \in S'$ if $s \in S$ or $R''(s)$, otherwise.

Definition 5. The firing rule for PNMIs is, then, as follows. A marking M of a PNMI leads to a marking M' by firing a compound transition T iff **(1)** there exists a subgraph S of M in which T is enabled under role R, **(2)** there exists a graph isomorphism i from M to M' **(3)** for every node $s \in S$ the labels $(name, M_s)$ in M and $(name', M_{i(s)})$ in M' are such that $R(s) = name.act$, $name = name'$ and M_s lead to $M_{i(s)}$ by firing a transition labelled with act and **(4)** for every node $s \notin S$, the corresponding labels in M and M' are the same.

2.2. Recall of DEVS models

Given a set of input events E_I and a set of output events E_O, a DEVS atomic model can be defined as a 9-tuple $(I_P, X, O_P, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta)$, where I_P is a set of input ports, $X \subseteq I_P \times E_I$ is the set of input ports and events, O_P is a set of output ports and $Y \subseteq O_P \times E_O$ is the set of output ports and events, S is the states set, Y is the output events set, $\delta_{int} : S \to S$ is the internal transition function, and $\delta_{ext} : S \times \mathbb{R}_{\geq 0} \times X \to S$ is the external transition function that returns the next state of the system, given the actual state, the amount of time since the actual state was reached, and the actual state. $\lambda : S \to Y$ is an output function that, given the actual state, returns the output event to communicate when an internal transition is performed. ta is a function $ta : S \to \mathbb{R}_{\geq 0} \cup \{\infty\}$. If the system is in state s, an internal transition is performed $ta(s)$ units of time after s is reached.

DEVS coupled models can be described by 6-tuple

$(X, Y, D, \{M_i\}, C, \mathsf{select})$, where X and Y are as before. D is a set of model indices. For every $i \in D$, M_i is a DEVS atomic model. The set $\{M_i\}$ must be such that no port belongs to two models. If I_P (O_P, respectively) denotes all the input ports (output ports, respectively) in the models M_i, $C \subseteq I_P \times O_P \cup O_P \times I_P$ is a coupling specification such that $(p_1, p_2) \in C$ implies that p_1 and p_2 belong to different models. We do not define external nor internal coupling, since this definition suffices for our purposes. We can suppose the input ports (output ports, resp.) of the coupled model are the input ports not connected to an output port (input port, resp.). The function $\mathsf{select} : \mathbb{P}D \setminus \emptyset \to D$ is used to choose the first atomic model to execute its internal transition, if there are many models having scheduled their internal transition for the same time.

In many cases it is useful to avoid resetting the remaining time until the next internal transition when an external event happens. In our encoding, we will use a trick already used in (Cristiá, 2007). The amount of time until the next internal transition is kept as part of the state and at every ocurrence of an external event this value is updated. If the states of the system are given by the set K, the set of states of the DEVS model will be $K \times (\mathbb{R}_{\geq 0} \cup \{\infty\})$. So, we define $\delta_{ext}((s, \sigma), t, e) = (s, \sigma - t)$. By doing so, the ta function becomes simply $ta(s, \sigma) = \sigma$.

3. EXTENDING PNMIS WITH TIME CONSTRAINTS

As explained in the introduction, time is a crucial aspect in DEVS. So, we extend the PNMIs in order to handle timing constraints.

We do so by allowing to specify that a token can be used to fire a transition only if a given amount of time has passed since the token was created. We show a very simple example in order to show how this kind of restrictions can be used in real-life systems. Consider the Petri net in figure 5.

Instead of black dots, we represent each token using a number: the specific number indicates the amount of time elapsed since the token was fired. For each arc pointing to a transition, we specify the amount of time that the tokens fired to the place need in order to become available for the transition. At the beginning of the sequence shown in the example, we have a token ready to be fired (the amount of time units needed to be available defaults to 0 if there are no indications in the arcs). Next, a transition is fired and we have two tokens in two different places. As the numbers above and below the places indicate, a token needs 3 units of time, while the other one needs 5 units. This marking leads to a marking in which both tokens have 3 time units. In this latter marking, a token is ready to be fired, while the other one still needs 2 units. But this marking leads to a marking in which both tokens have 5 units (not shown in the figure), and then the tokens are ready to be fired, producing a token in the rightmost place (as shown in the figure).

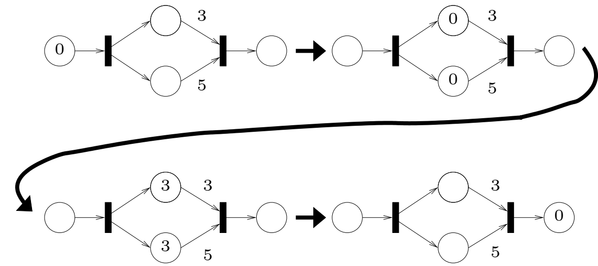

Figure 5. A simple example of a timed PNMI

The nets in figure 5 represents an AND-split followed by an AND-join (van der Aalst and van Hee, 2002): the places in the middle represent tasks demanding some time. As time passes, one of the task is finished, but the other one is still being performed. At the end, both tasks are finished, and so the compound task is.

In general, a marking M can lead to a marking M' if either the number corresponding to each token is increased by the same amount t (we say that M leads to M' through a step of length t), or if a transition T is executed (as before, we say that M leads to M' through transition T). The execution of a transition is supposed to be instantaneous. That is, the number corresponding to the tokens is not changed when a transition occurs. So, transitions model changes, not tasks. The performance of a task is indicated by the presence of a token in a place. So, the timing restrictions introduced are useful to model the amount of time needed by the different tasks performed in the system.

So far, we did not restrict the set from which the constants for the timing constraints are taken. However, we did not find any issue concerning this particular set. (Note that we are dealing with a fixed number of instances, and the number of constraints is finite.) So, we can simply consider that all the timing constraints are nonnegative real numbers and all our definitions are still consistent.

Note that, if the new firing rule for timed PNMIs had to be expressed formally, the hardest part of the definition (relative to the selection of a set of instances in order to fire a compound transition) remains unchanged. The changes to definitions 1, 2 and 3 are not difficult and they are not shown here. With respect to definition 5, we write a new firing rule for timed PNMIs.

Definition 6 (Firing rule for timed PNMIs)**.** A marking M of a timed PNMI leads to a marking M' through a step of length t if there exists a graph isomorphism i from M to M' such that, for every node s in M, the labels $(name, M_s)$ in M and $(name', M_{i(s)})$ in M' are such that $name = name'$ and M_s leads to $M_{i(s)}$ through a step of length t. The part of definition dealing with changes made by firing a compound transition is exactly definition 5.

4. AN ILLUSTRATIVE EXAMPLE USING TIMED PNMIS

In order to explain our encoding using a concrete example, we present a PNMI modelling the production of an item. This example serves both to provide a concrete example illustrating our encoding and to show the expressive power of timed PNMIs.

Suppose that an item must be produced according to a given workflow specification. In addition, some resources must be used in order to perform the tasks as required by the workflow specification. Then, simulations may result useful to try different plans for the tasks to perform, as well as schedules for the resources needed.

In our concrete example, we have an item that requires two tasks $T1$ and $T2$ in order to be completed. These tasks can be performed in any order, and even simultaneously. (Note that, although we are using a simple workflow, the workflow for the example could be arbitrarily complex.) The items are disposed on different conveyor belts, where robotic arms perform the needed tasks. Because of restrictions inherent to the use of arms in the belts, at most two arms are allowed to be assigned to the same belt at the same time. The movement of an arm from a belt to another takes 4 time units. The task $T1$ takes 2 time units, and the task $T2$ takes 7 time units. Once an item has been put on a belt, it takes 3 units to reach the point where the arms can perform the tasks. A belt cannot be moved if some arm is performing a task in that belt. It is worth noting that the example illustrate how to specify restrictions on the order in which tasks are performed (using the workflow specification), restrictions on the usage of resources (at most two arms may be assigned to the same belt) and timing constraints.

The model we present is intended to try different production schedules (for instance, "put an item on each belt, send one arm to each belt in order to perform T1, ...") and evaluate the performance of the system using these schedules. This model is not intended to verify that the belt will appropriately stop when needed nor that the arms perform the required tasks on time.

We define 5 Petri net models called: Arm, Belt, Item, ArmBelt and ItemBelt. The former three models specify the expected behaviour of the entities of the system. The latter two modules model fictitious entities representing the relationships among the other models: for example, if an arm is assigned to a belt, or if an item is put on a belt. This modelling technique resembles the entity-relationship diagrams used to design databases.

In order to keep the information concerning to all of the possible relationships, for every instance of type Arm and every instance of type Belt, an instance of type ArmBelt exists corresponding to the relationship between the particular Arm and the particular Belt, and similarly for every Item and every Belt. The hierarchy for the case in which the system is composed of two arms, two belts and two items is depicted in figure 6.

Next, we must define the models. They are shown in figure 7. In order to improve readability, we do not show the black bar when a transition has only one input place and an output place, we simply draw an arc from the input place to the output place with the corresponding label. Since the weights for all arcs are 1, we omit them. So, the numbers in the arcs correspond to timing constraints.

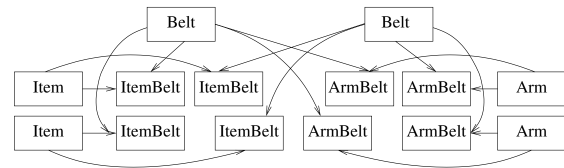

Figure 6. The hierarchy for the example

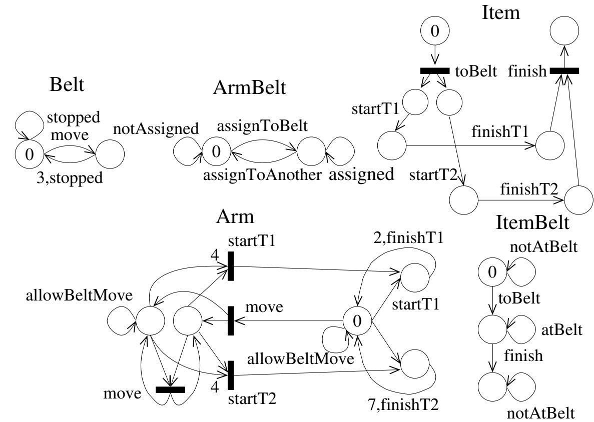

Figure 7. A non-trivial example of a timed PNMI

The model for the belts specifies that, when stopped, the belt can be moved. Then, after 3 time units, the belt stops, since the item is in the right place. Once the belt has stopped, the action $stopped$ can be executed. In some compound transitions, the action $stopped$ is executed in order to ensure that the belt is stopped. We often use actions in order to require an instance to be in a certain state. For instance, note that the states in which the belt is stopped are both the states in which a token is present in the leftmost place and the states in which a token created more than 3 time units ago is present in the rightmost place. That is, such states are precisely the ones in which the action $stopped$ is enabled.

The model ArmBelt specifies that a given arm may be assigned to a given belt or not in a particular moment of the execution. If an arm is assigned to a belt, the instance of ArmBelt corresponding to the particular arm and the particular belt will be able to execute the transition $assigned$. Otherwise, the instance will be able to execute the transition $notAssigned$.

The model for the item specifies that an item is put on the belt and then $T1$ and $T2$ are performed in any order. The item is finished when both $T1$ and $T2$ are finished. Note that the timing constraints are not in the model of the item. The model of the item is a workflow specifying the possible production plans yielding a finished item.

The model ItemBelt specifies that initially an item is not assigned to a belt unless the action $toBelt$ has been executed. The item is at the belt until the action $finish$ is executed.

For the model of the arm, we suppose that each arm is initially ready to work in a belt, but not working. So, the belt can be moved. This ability is reflected by the enabledness of the action $allowBeltMove$. In the initial state, the arm can start either $T1$ or $T2$. Once a task has been started, the action $allowBeltMove$ is not enabled any more. It gets enabled again once either $T1$ or $T2$ have finished. In the initial state, the arm can also be asked to move. In this case, tokens are fired to two places. One of the tokens is used to fire the $allowBeltMove$ action if needed, the other one is used to save the amount of time elapsed since the firing of $move$. Note that the execution of $move$ resets the elapsed time, while the execution of $allowBeltMove$ does not.

The compound transitions for the PNMI are shown in figure 8.

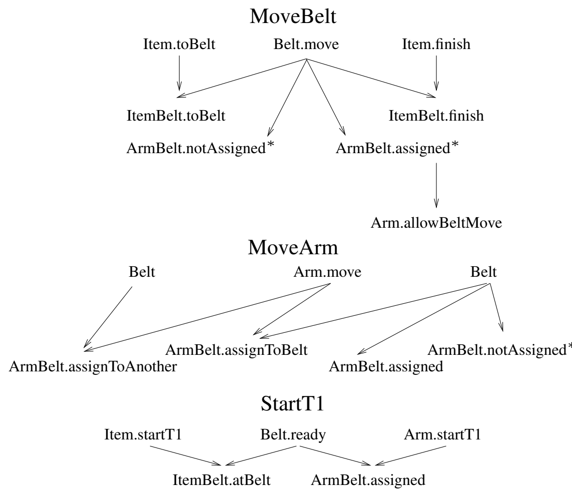

Figure 8. Compound transitions for the example

The compound transition MoveBelt needs two items: one item that is already finished and another one that will be put in the belt. A similar transition (not shown in the figure, including a node $ItemBelt.notAtBelt^*$) may allow the movement if there is no finished item. For the finished item, proper actions are invoked both for the item itself and for the corresponding instance of type ItemBelt, and similarly for the unprocessed item. Note that the arms not assigned to the belt are mapped to $ArmBelt.notAssigned^*$, while the instances of type ArmBelt corresponding to arms assigned to the belt being moved must be able to execute $allowBeltMove$.

For the compound transition MoveArm, we add a bit of "syntactic sugar" to the PNMIs. Here, we have to consider two belts (the belt that the arm is assigned to before the transition, and the one it is assigned to after the transition), but no actions need to be performed for these belts. In general, an instance may be referred only because of its place in the hierarchy, but no action needs to be performed. In this case, we simply write the type of the instance. To see that this modification can be encoded without adding expressive power to the PNMIs, we may suppose that every instance has a "bogus" action which is enabled in all the states. If the instance has timing constraints, the trick we used in order to avoid resetting the elapsed time for the action allowBeltMove can be used for the bogus action. Then, $Belt$ is simply syntactic sugar for $Belt.bogus$.

All that said, the compound transition MoveArm requires two belts. In the transition shown in the figure, exactly one arm must be assigned to the destination belt. A similar transition (obtaining by merely erasing the $ArmBelt.assigned$ node) represents the movement when there are no arms assigned to the destination belt. Note that we execute $assignToBelt$ in the instance of type $ArmBelt$ corresponding to the destination belt and $assignToAnother$ in the instance corresponding to the source belt.

The transition StartT1 can be read as follows: if there is an arm, a belt and an item such that the arm is in the same belt as the item, and the arm and the item can start $T1$, then $T1$ can be started.

5. FROM TIMED PNMIS TO DEVS

5.1. Our encoding

In this section, we show how to encode timed PNMIs into DEVS models.

In the encoding we present, each (timed) Petri net N in the timed PNMI is mapped to an atomic DEVS module. A fictitious model named *coordinator* receives instructions to execute compound transitions according to role functions. If the compound transition can be performed, the coordinator indirectly asks every atomic DEVS module involved in the transition to execute the corresponding action. Fictitous "translator" models are defined for each instance. The translators receive the compound transition to execute and a role function. Once a compound transition is received, the translators communicate the action to be performed to the model corresponding to the instance. This part of the encoding is depicted in figure 9, for the compound transition StartT1 (the instances of type ArmBelt and ItemBelt are not shown).

In order to decide whether a compound transition can be executed or not, the coordinator must keep the information relative to the enabledness of actions in all the instances. So, every time an action gets enabled/disabled, the model corresponding to the instance communicates the new set of enabled actions to the coordinator.

Each instance i in the timed PNMI is mapped to an atomic DEVS module as follows: the set of states S are of the form (M, σ), where M is a marking for i and $\sigma \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ is used to save the amount of time until the next internal transition (so, we define $ta(M, \sigma) = \sigma$).

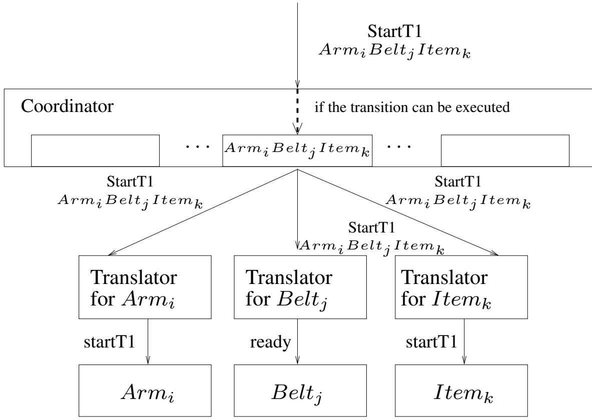

Figure 9. Encoding a compound transition

Each atomic model has only one input port i_i and one output port o_i. The input events are of the form (i_i, act), where act is the name of an action in i. So, the ocurrence of the event act means that the model is asked to perform the action act. The δ_{ext} function is then defined by $\delta_{ext}((M, \sigma), t, (i_i, act)) = (M', 0)$, where M' is the (univocally defined, since the Petri nets are assumed to be deterministic on actions) marking obtained by executing a step of length t and then executing the action act in M. We set the time until the next internal transition to 0, since we need to force an internal transition each time an event happens, in order to communicate the new set of enabled actions to the coordinator.

The atomic models have one output port called o_i. This output port is used to communicate the set of actions that the Petri net is allowed to execute. So, the output events are of the form $(o_i, \{act_1, \ldots, act_n\})$. In DEVS models, an output is produced each time an internal transition happens. Since each output is a set of enabled outputs, an internal transition should happen every time an action becomes enabled/disabled. The enabledness of actions may change because of the passage of time, or because of the execution of a transition. Consider the belt shown in the example in the previous section. Suppose that the belt is initially stopped. Once the action $move$ has been executed, we must communicate that there are no available actions. Once the set of enabled actions is communicated, we must set the time until the next internal transition to be the time until an action becomes enabled because of the passage of time.

In the case of the belt, after communicating that no actions are enabled, we must set the ta function to 3. Since $ta(M, \sigma) = \sigma$, we do so by choosing a state of the form $(M, 3)$. Hence, after 3 units of time, the output $\lambda((M, 3)) = \{stopped\}$ is produced if no event happens. Notice that, since the parameter of λ is the state *before* the transition is performed, $stopped$ is *not* enabled in M. In general, the output issued because of the passage of time is the set of actions enabled in the *next* state. However, immediately after the action $move$, the output $\lambda((M, 0)) = \{\}$ was executed, for the same M. So, the output issued because of the execution of an action is the set of actions enabled in the *actual* state. Since the outputs issued because of the execution of an action are executed when the second component of the pair is 0, and the outputs issued by the passage of time are executed only when the second component of the pair is not 0 (since the amount of time until the next change of the set of available actions is always greater than 0), we used the second component of the pair in order to separate both cases. Hence, $\lambda((M, 0)) = (o_i, \{act | act \text{ is enabled in } M\})$ and $\lambda((M, t)) = (o_i, \{act | act \text{ is enabled in } M'\})$, where M' is the marking obtained by taking a step of length t in M.

We define the δ_{int} function as follows: if $t \neq \infty$, then $\delta_{int}((M, t)) = (M', t')$, where M' is the marking obtained by taking a step of length t in M and t' is the minimum amount such that after t' time units the set of available actions in M' changes. If no such t' exists, $\delta_{int}((M, t)) = (M', \infty)$.

So, we defined DEVS models corresponding to each type of instance. We need now to coordinate the actions of the individual models. We use a fictitious model, called "coordinator", that is asked to perform compound transitions. The coordinator perform the compound transitions by sending outputs to the corresponding instances. It receives information about the available actions in each instance.

The coordinator has one input port t_C receiving compound transitions, it has M input ports a_C^i receiving sets of actions (where M is the amount of instances) and it has one output port o_C^S for each subset S of instances. In the following, we will assume that a pair (T, r) is such that T is the name of a compound transition and r is a role function for this compound transition. Such role functions must honor the hierarchy, but they may or may not honor the enabledness of the labels in the compound transition T, depending on the states of the atomic models corresponding to the instances. The input events are either of the form $(t_C, (T, r))$, or of the form $(a_C^i, \{act_1, \cdots, act_n\})$. The output events are of the form $(o_C^S, (T, r))$, where S is the domain of r.

The coordinator only needs to know what pairs (T, r) are feasible, given the actions enabled in each instance. Hence, we include in the state of the coordinator a function $en : I \to \mathbb{P}\mathcal{A}$, where I is the set of instances and $\mathcal{A}$ is the set of all the actions. In addition, each state has a value which is either the value $none$ or it is of the form (T, r). This value is used to distinguish the states in which the coordinator is going to execute a transition. When the coordinator is asked to perform a compound transition, it must check that that the transition can be executed.

Hence, we define $\delta_{ext}(\,(en, none)\,,\,t\,,\,(t_C,(T,r))\,) = (en,(T,r))$, if the domain of the role function is a subgraph for which T is enabled according to en. Otherwise, $\delta_{ext}(\,(en, none)\,,\,t\,,\,(t_C,(T,r))\,) = (en, none)$. In the case in which the first parameter is $(en,(T,r))$ for some (T,r), we leave the function undefined, since in this case $ta(en,(T,r)) = 0$ and so no external events can happen while being in this state. So, $ta(\,(en,\,(T,r)\,)\,) = 0$, $ta(\,(en, none)\,) = \infty$. So far, δ_{ext} is not completely defined, since we need to specify how the coordinator changes its state according to the changes relative to the enabledness of the actions. Fortunately, this case is simple: $\delta_{ext}(\,(en, none)\,,\,t\,,\,(a^i_C,\{act_1,\cdots,act_n\})\,) = (en', none)$, where $en'(j) = \{act_1,\cdots,act_n\}$ if $j = i$, or $en'(j) = en(j)$, otherwise.

The output function is $\lambda(\,(en,(T,r))\,) = (\,o^S_C,(T,r)\,)$, where S is the domain of r. The δ_{int} function is simply $\delta_{int}(\,(en,\,(T,r)\,)\,) = (en, none)$. That is, once the output is produced, there is nothing to execute.

Note that the outputs of the coordinator consist of pairs (T,r), while the inputs of the models corresponding to the instances consist of actions. In order to overcome this difficulty, we add a "translator" for each instance. The translator simply receives a pair (T,r) and outputs the action corresponding to the instance. This difficulty cannot be avoided, unless the coordinator is defined to perform multiple outputs for each request. We wanted the coordinator to be as simple as possible, and so we defined it to perform a single output for each request. So, this output must include all the information. A dirty way to avoid the difficulty is to change the inputs of the models corresponding to the instances, defining the inputs to be of the form (T,r), but this change severely affects modularity. In our solution, although the translators are related to instances and receive information concerning the whole PNMI, the lack of modularity is isolated in simple components.

It is worth also mentioning that we looked for the more "elegant" encoding. In practice, generating all the output ports o^S_C may be very inadequate. We did so to avoid sending outputs to instances not involved in the compound transition. However, the outputs could be defined to be sent to all the translators, and the translators could be defined to discard the input if the instance corresponding to the translator is not in the domain of r. This latter approach should be used if the overhead caused by irrelevant messages does not compromise the performance of the simulation, and if the ports are generated statically and they consume computer resources.

The translator for an instance i has one input port i_{Z_i} and one output port o_{Z_i}. Input events are of the form $(i_{Z_i},\,(T,r)\,)$, and output events are actions sent to the port o_{Z_i}. The states are $\mathcal{A}_i \cup \{none\}$, where $\mathcal{A}_i$ are the actions in the instance i. ta is defined by $ta(none) = \infty$ and $ta(act) = 0$. To complete the definition, let's define $\lambda(act) = act$, $\delta_{int}(act) = none$, $\delta_{ext}(none, t, (T,r)) = act$ where $r(i) = n$ and n is a node in T whose label is of the form $name.act\,[\,^*\,]$.

Next, we describe the pairs in the coupling specification C. For each subset S of instances, the pairs in the set $\{(o^S_C, i_{Z_i}) | i \in S\}$ are in C. C also includes all the pairs $\{(o_{Z_i}, i_i)\}$ and $\{(o_i, a^i_C)\}$ for every instance i.

Finally, we define the select function. In the following, we assume that no external events can occur at the same time and, furthermore, we assume that no external events can occur while an atomic model is in a state s such that $ta(s) = 0$. Quoting (Zeigler *et al.*, 2000): "[When $ta(s) = 0$] the stay in the state s is so short that no external events can intervene —we say that s is a transitory state." Under these assumptions, all the communications depicted in figure 9 and the communications of the new sets of enabled actions after an external event occur *before* the next external event. We define select to choose the coordinator over all the other models. Note that another model may be trying to execute an internal transition only if the model corresponds to an instance for which the enabledness of actions needs to be changed because of the passage of time. So, in this case, if a compound transition is asked to be performed *exactly* when the enabledness of the actions changes, either the actual or the next enabledness may be considered depending on whether the external or the internal transition is chosen to be executed in the first place. For the same reason, translators must be selected must be chosen over instances. Since the executions of different translators (instances, respectively) are independent, the choice of the particular translator (instance, respectively) is irrelevant.

5.2 Zeno runs

Now we show an important property of the DEVS we obtain when starting from PNMIs. We define a Zeno run as an execution of a DEVS model in which infinite transitions are performed in a finite amount of time. A sequence of external events is a (possibly infinite) sequence of pairs (e,t), where e is an external event for the DEVS model under consideration and t is a real number indicating the time in which the event occurs (the sequence of t's must be strictly ascending). We say that a sequence of external events is non-Zeno if either it is finite or no subsequence

of events occurs in a finite amount of time. Usually, the sequences of external events for DEVS are assumed to be non-Zeno. Now, we are ready to state the theorem.

Theorem (DEVS encoding PNMIs have no Zeno runs)**.** *For every non-Zeno sequence of external events and every DEVS model obtained by using the encoding previously described, the execution of the DEVS model corresponding to the sequence of external events is non-Zeno.*

6. CONCLUSIONS AND FURTHER WORK

In order to obtain DEVS models, we extended PNMIs with timing constraints, and showed how the extended formalism can be used to model the interaction among production items, resources and tasks. In particular, the process used to produce an item is specified using a workflow. This suggests that the workflow corresponding to the production of an item may be developed and verified independently of the Petri nets corresponding to the resources. If the verifications corresponding to the items are successful, then the verification of the entire system can be attempted. Hence, this ability to separate the design of the items from the rest of the system avoids unnecesary verifications of the complete model.

Our encoding can be used to complement the formal verification of PNMI models. For instance, in the example in section 4, the property stating that no more than two arms are assigned to the same belt at the same time should hold independently of the timing constraints. Such properties could be proven using (less expensive) verification techniques for untimed systems, while the properties depending on timing constraints can be tested using simulation. Then, if necessary, the complete formal verification can be attempted once these steps yielded satisfactory results.

In our extension, we imposed only lower bounds to the time needed by the tasks (that is, we can specify that a token "needs at least two units"). Although upper bounds are needed in some cases, only the definition of the underlying Petri nets needs to be modified in order to take into account such restrictions. Note that, with this extension, the passage of time may imply the *disabling* of actions but the encoding needs not to be modified, since each time action gets enabled/disabled the new set of enabled actions is communicated.

We explained why the extension of the formalism with timing constraints only affects the definition of the underlying Petri nets, while the part corresponding to the coordination of multiple instances remains almost the same (excepting for the distinction among steps or transitions in the firing rule). It would be unfair to say that this fact shows the robustness of the formalism with respect to possible extensions, since the definition has changed with respect to previous versions. However, we developed the modifications to the formalism before we attempted the extension to timing constraints. The PNMIs presented in (Giro and Frydman, 2006) were defined trying to avoid a separate specification for the actions in which different instances interact. In this context, the set of instances required to perform a transition is derived from the structure of the network. The reason for such an avoidance was to allow only a set predefined interactions, thus restricting computational expresiveness in order to make the reachability problem more tractable. Nevertheless, having the specification of the different instances required to interact embedded into the whole specification yields quite complex definitions. Here, we found an interesting compromise: the interactions are still constrained, and the formalism is now simpler. Here, we also restrict the set of instances to be finite, since the PNMIs are now intended for simulation. For the same reason, we do not allow the creation of new instances (all the instances to be simulated can be created beforehand and "activated" using a special action). It would be interesting to extend PNMIs with the ability to specify other aspects (for instance, probabilities) in order to see how much the formalism must be modified to specify new aspects.

In the future, we plan to focus on verification of PNMIs. Although finite-state PNMIs can be seen as Kripke structures and verified using model-checking techniques as the ones presented in (Clarke *et al.*, 1999), we aim to develop methods for automatic verification exploiting the particular structure of the PNMIs.

REFERENCES

van der Aalst, W. M. P. and K. M. van Hee, 2002. *Workflow Management: Models, Methods, and Systems.* MIT Press.

van der Aalst, W. M. P. and A. H. M. ter Hofstede, 2005. YAWL: yet another workflow language. *Information systems*, 30(4), p. 245–275.

Clarke, E. M., O. Grumberg, D. A. Peled, 1999. *Model Checking.* MIT Press.

Cristiá, M., 2007. A TLA+ Encoding of DEVS Models. *Proceedings of the International Modelling and Simulation Multiconference (IMSM 2007)*, Buenos Aires.

Giro, S., 2007. Workflow Verification: a New Tower of Babel. *Proceedings of the International Modelling and Simulation Multiconference (IMSM 2007).*

Giro, S. and C. Frydman, 2006. Modelling Workflows using Petri nets with Multiple Instances. *Proceeding of the Argentine Symposium on Computing Technology (AST 2006)*, Mendoza, Argentina.

Valiente, G., 2002. *Algorithms on trees and graphs.* Springer.

Zacharewicz, G., *Un environnement G-DEVS/HLA: application a la modélisation et simulation distribuée de workflow*, PhD. Thesis, U. Aix-Marseille III, 2006.

Zeigler, B. P., H. Prähofer and T. G. Kim, 2000. *Theory of Modeling and Simulation.* Academic Press.