# Extending PNML Scope: a Framework to Combine Petri Nets Types

L.M. Hillah[1], F. Kordon[2], C. Lakos[3], and L. Petrucci[4]

[1] LIP6, CNRS UMR 7606 and Université Paris Ouest Nanterre La Défense
200, avenue de la République, F-92001 Nanterre Cedex, France
Lom-Messan.Hillah@lip6.fr
[2] LIP6 - CNRS UMR 7606, Université P. & M. Curie
4 Place Jussieu, F-75252 Paris cedex 05, France
Fabrice.Kordon@lip6.fr
[3] University of Adelaide, Adelaide, SA 5005, Australia
Charles.Lakos@adelaide.edu.au
[4] LIPN, CNRS UMR 7030, Université Paris XIII
99, avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France
Laure.Petrucci@lipn.univ-paris13.fr

**Abstract.** The Petri net standard ISO/IEC 15909 comprises 3 parts. The first one defines the most used net types, the second an interchange format for these – both are published. The third part deals with Petri net extensions, in particular structuring mechanisms and the introduction of additional, more elaborate net types within the standard.
This paper presents a contribution to elaborate an extension framework for the third part of the standard. This strategy aims at composing enabling rules and augmenting constraints in order to build new Petri net types. We show as a proof of concept how this can be achieved with priorities, times, inhibitor arcs in the context of an interleaving semantics. We then map this framework onto the current standard metamodels.

**Keywords:** Standardisation, PNML, Prioritised Petri Nets, Time Nets

## 1 Introduction

**Context** The International Standard on Petri nets, ISO/IEC 15909, comprises three parts. The first one (ISO/IEC 15909-1) deals with basic definitions of several Petri net types: Place/Transition, Symmetric, and High-level nets[5]. It was published in December 2004 [13]. The second part, ISO/IEC 15909-2, defines the interchange format for Petri net models: the Petri Net Markup Language [15] (PNML, an XML-based representation). This part of the standard was published on February 2011 [14]. It can now be used by tool developers in the Petri Nets community with, for example, the companion tool to the standard, PNML Framework [11].

The standardisation effort is now focussed on the third part. ISO/IEC 15909-3 aims at defining extensions on the whole family of Petri nets. Extensions are, for instance,

[5] In this paper, the term "high-level net" is used in the sense of the standard and corresponds to coloured Petri nets as in Jensen's work [16].

the support of modularity, time, priorities or probabilities. Enrichments consider less significant semantic changes such as inhibitor arcs, capacity places, etc. This raises flexibility and compatibility issues in the standard.

While parts 1 and 2 of the ISO/IEC 15909 standard address simple and common Petri nets types, part 3 is concerned with extensions. The work on these issues started with a one-year study group drawing conclusions with respect to the scope to be addressed. Then, according to the study group conclusions, the standardisation project was launched in May 2011, for delivery within 5 years.

**Contribution** The choices to be made in part 3 of the standard must of course ensure compatibility with the previous parts. We propose to achieve this goal by using the notion of *orthogonality*. It allows us to build a *framework* to describe the behavioural semantics of nets in a compositional way. This is achieved by revisiting the firing rule of several well known Petri net types based on the enabling rule. The objective is to compose existing enabling rules with augmenting constraints in order to elaborate these Petri net types consistently. We apply it to P/T nets, Prioritised nets and Time nets.

Associated with this framework is a MDE-based one that allows us to compose pieces of metamodels corresponding to *enabling functions* and *augmenting constraints* (see section 3.2). This is an important link between the syntax-based way of handling semantic in MDE techniques and the formal definition of a behavioural semantics.

Based on this framework, we map the selected Petri nets types onto our MDE-based framework as a proof of concept, as planned for in future evolutions of the standard.

**Content** Section 2 recalls some well-known Petri nets types. Then, section 3 presents the framework to define the behavioural semantics of nets and applies it to the net types already presented. Section 4 details the MDE-based framework associated with the standard. Section 5 maps the notions of the behavioral semantics defined in section 3 into this MDE-based framework to build metamodels of net types suitable for generating PNML descriptions, followed by a discussion in section 6.

## 2 Some Petri Nets Definitions

This section introduces the notations for different types of Petri Nets.

### 2.1 Definition of Place/Transition Nets

This section first introduces Place/Transitions nets.

**Definition 1 (Place/Transition Net).**
*A* Place/Transition Net *(P/T net) is defined by a tuple* $N = \langle P, T, Pre, Post, M_0 \rangle$*, where:*

- $P$ *is a finite set (the set of places of* $N$*),*
- $T$ *is a finite set (the set of transitions of* $N$*), disjoint from* $P$*,*
- $Pre, Post \in \mathbb{N}^{|P| \times |T|}$ *are matrices (the backward and forward incidence matrices),*
- $M_0$*, a vector in* $\mathbb{N}^{|P|}$ *defining the initial number of tokens in places.*

We now introduce $M(p)$, $\bullet t$, and $t\bullet$ that are respectively:

- the current marking of place $p$,
- the subset of places which constitute the precondition of a transition $t \in T$,
- the subset of places which constitute the postcondition of a transition $t \in T$.

From these notations, we can define the enabling and firing rules for P/T nets as follows.

**Definition 2 (P/T Net enabling rule).**
*A transition $t \in T$ is* enabled *in marking $M$, denoted by $M[t\rangle$, iff: $\forall p \in \bullet t, M(p) \geqslant Pre(p,t)$.*

**Definition 3 (P/T Net firing rule).**
*If a transition $t \in T$ is* enabled *in marking $M$, it can* fire *leading to marking $M'$, denoted by $M[t\rangle M'$, where: $\forall p \in P, M'(p) = M(p) - Pre(p,t) + Post(p,t)$.*

An *inhibitor arc* is a special kind of arc that reverses the logic of an input place. Instead of testing the presence of a minimum number of tokens in the related place, it tests the lack of tokens.

**Definition 4 (Petri Nets with Inhibitor Arcs).**
*A Petri net with inhibitor arcs is a Petri Net $N$ together with a matrix $I \in \mathbb{N}^{|P|\times|T|}$ of inhibitor arcs.*

**Definition 5 (P/T Nets with inhibitor arcs enabling rule).**
*A transition $t \in T$ is* enabled *in marking $M$, denoted by $M[t\rangle$, iff: $\forall p \in \bullet t, (M(p) \geqslant Pre(p,t)) \wedge (M(p) \leqslant I(p,t))$.*

Then, the firing rule is identical to the one for P/T nets, provided the transition is enabled.

### 2.2 Definition of Prioritised Petri Nets

This section introduces the definition of prioritised Petri nets.

**Definition 6 (Statically Prioritised Petri net).**
*A* Statically Prioritised Petri net *is a tuple $SPPN = \langle P, T, Pre, Post, M_0, \rho\rangle$, where:*

- *$\langle P, T, Pre, Post, M_0\rangle$ is a P/T net.*
- *$\rho$ is the* static priority function *mapping a transition into $\mathbb{R}^+$.*

We can also consider the case where the priority of transitions is *dynamic*, i.e. it depends on the current marking [1]. This definition was introduced in [19]. Note that the only difference with statically prioritised Petri nets concerns the priority function $\rho$.

**Definition 7 (Prioritised Petri net).**
*A* Prioritised Petri net *is a tuple $PPN = \langle P, T, Pre, Post, M_0, \rho\rangle$, where:*

- *$\langle P, T, Pre, Post, M_0\rangle$ is a P/T net.*

– $\rho$ *is the* priority function *mapping a marking and a transition into* $\mathbb{R}^+$.

The behaviour of a prioritised Petri net is now detailed, markings being those of the underlying Petri net. Note that the firing rule is the same as for non-prioritised Petri nets, the priority scheme influencing only the enabling condition.

**Definition 8 (Prioritised enabling rule).**
*A transition* $t \in T$ *is* priority enabled *in marking* $M$*, denoted by* $M[t\rangle^\rho$*, iff:*

– *it is enabled, i.e.* $M[t\rangle$*, and*
– *no transition of higher priority is enabled, i.e.* $\forall t' : M[t'\rangle \Rightarrow \rho(M,t) \geq \rho(M,t')$.

*The definition of the priority function* $\rho$ *is extended to sets and sequences of transitions (and even markings* $M$*):*

– $\forall X \subseteq T : \rho(M,X) = \max\{\rho(M,t) \mid t \in X \wedge M[t\rangle\}$
– $\forall \sigma \in T^* : \rho(M,\sigma) = \min\{\rho(M',t') \mid M'[t'\rangle^\rho$ *occurs in* $M[\sigma\rangle^\rho\}$.

For static prioritised nets where $\rho(M,t)$ is a constant function associated with $t$ and for dynamic prioritised nets, for at least one transition, $\rho(M,t)$ depends on the current marking. For now, we will consider only dynamically prioritised nets since static ones are encompassed by these.

If the priority function is constantly zero over all markings and all transitions, then the behaviour of a Prioritised Petri Net is isomorphic to that of the underlying P/T Net.

Note that we choose to define priority as a positive real-valued function over markings and transitions — the higher the value, the greater the priority. We could equally define priority in terms of a *rank function* which maps markings and transitions to positive real values, but where the smaller value has the higher priority. This would be appropriate, for example, if the rank were an indication of earliest firing time.

### 2.3 Definition of Petri Nets with Time

*Time Petri nets* (TPN) are Petri nets where timing constraints are associated with the nodes or arcs. Timing constraints are given as time intervals. This section briefly presents the definition of TPNs and their semantics [3], then introduces the model we will be focusing on, which is *Time Petri Nets* [4], where the timing constraints are associated with transitions.

**Definition 9 (Generic Time Petri net).**
*A* Generic Time Petri net *is a tuple* $\langle P, T, Pre, Post, S_0, I\rangle$ *such that:*

– $\langle P, T, Pre, Post, S_0\rangle$ *is a (marked) P/T net (*$S_0$ *denotes its initial state, i.e. marking + clocks);*
– $I : X \to \mathcal{I}(\mathbb{R}_+)$ *is a mapping from* $X \in \{P, T, P \times T \cup T \times P\}$ *to the set* $\mathcal{I}(\mathbb{R}_+)$ *of intervals. These intervals have real bounds or are right-open to infinity.*

**Semantics** The semantics of TPNs is based on the notion of *clocks*. One or more clocks can be associated with a time interval and the value of all clocks progress synchronously as time elapses. The firability of enabled transitions depends on having the value of the related clocks in their associated intervals. A clock may be reset upon meeting a condition on the marking of the net, usually after the firing of a transition.

The semantics of TPNs is defined in terms of:

- *Reset policy*: the value of a clock is reset upon firing some transition. It is the only way to decrease its value. But it is also meaningful not to reset a clock.
- *Strong firing policy*: in a *strong* semantics, when the upper bound of the interval associated with a clock is reached, transitions must fire instantaneously, until the clock is reset. The clock can go beyond the upper bound of the interval, if there is no possible instantaneous sequence of firings, in which case dead tokens are usually generated. This generally models a bad behaviour, since tokens become too old to satisfy the timing constraints.
- *Weak firing policy*: in this case, the clock leaving the interval prevents the associated firings from taking place. Dead tokens may also be generated, but this time they are considered part of the normal behaviour of the net.
- *Monoserver setting*: each interval only has one associated clock, which usually denotes a single task processing.
- *Multiserver setting*: each interval has more than one associated clock, which usually denotes the handling of several similar tasks. In this setting, each clock evolves independently of the others.

**Time Petri nets** We consider in this paper the Time Petri Net model [4], where time intervals are associated with transitions. The semantics is strong and monoserver. Time Petri Nets are appropriate for modelling real-time systems. More formally:

**Definition 10 (Time Petri net).** *A* Time Petri Net *is a tuple* $\langle P, T, Pre, Post, S_0, \alpha, \beta \rangle$ *where:*

- $\langle P, T, Pre, Post, S_0 \rangle$ *is a (marked) P/T net.*
- $\alpha : T \mapsto \mathbb{Q}_+$ *and* $\beta : T \mapsto \mathbb{Q}_+ \cup \{\infty\}$ *are functions satisfying* $\forall t \in T, \alpha(t) \leq \beta(t)$ *called respectively* earliest ($\alpha$) *and* latest ($\beta$) *transition firing times.*

Functions $\alpha$ and $\beta$ are the instantiation of $I$ in Generic Time Petri nets (see definition 9) for Time Petri nets.

Given a marking $M$, we write $En(M) = \{t \in T \mid M[t\rangle\}$ for the set of transitions enabled in $M$. A clock is implicitly associated with each transition and a state of the system is a pair $(M, v)$, where $M$ is a marking and $v \in \mathbb{R}_+^{En(M)}$ is a mapping associating a clock value with each transition enabled in $M$.

We now define the enabling rule and firing rule of Time Petri nets.

**Definition 11 (Time Petri net enabling rule).**
*A transition* $t \in T$ *is* time enabled *in state* $(M, v)$*, denoted by* $(M, v)[t\rangle$*, if:*

- *it is enabled, i.e.* $M[t\rangle$*, and*

– $v(t) \in [\alpha(t), \beta(t)]$.

**Definition 12 (Time Petri net firing rule).**
*From a state $(M, v)$, two types of transitions are possible:*

– *if transition $t \in T$ is* time enabled *in state $(M, v)$,* firing *$t$ leads to state $(M', v')$, denoted by $(M, v)[t\rangle(M', v')$, where:*
  • $\forall p \in P, M'(p) = M(p) - Pre(p, t) + Post(p, t)$*, as usual,*
  • $\forall t' \in En(M'), v'(t') = 0$ *if $t'$ is newly enabled (explained below), and $v(t')$ otherwise.*
– *if $\forall t \in En(M), v(t) + d \leq \beta(t)$,* time elapsing *by delay $d \in \mathbb{R}_+$ leads to state $(M, v')$, where $v'(t) = v(t) + d$ for all $t \in En(M)$.*

Various definitions have been proposed for newly enabling of a transition [2, 26]. A common one, called intermediate semantics, states that transition $t'$ is newly enabled by the firing of $t$ if:

– $t'$ belongs to $En(M')$ and
– either $t' = t$ or $t'$ is not enabled in $M - Pre(., t)$ (where $Pre(., t)$ denotes the vector $(Pre(p, t))_{p \in P}$).

## 3 An Engineering Approach to Extension and Composition

In considering extensions to the common Petri net types in parts 1 and 2 of the ISO/IEC 15909 standard, we wish to capture the extensions so that they are as flexible as possible, and hence applicable to multiple Petri net types. This is an engineering challenge like any software design — its success will be measured by a number of non-functional properties like extensibility, maintainability, usability and reusability.

In aiming for this goal, we wish to define extensions as "pieces of semantics" that are *orthogonal*. The term *orthogonal* has been applied in the literature to language design in a variety of ways depending on the context. We first review some of the literature on this notion before indicating how we propose to apply it in the standard.

### 3.1 The Notion of Orthogonality

**Orthogonality for programming languages** In the context of programming language design, Pratt and Zelkowitz put it this way [24]: *"The term* orthogonality *refers to the attribute of being able to combine various features of a language in all possible combinations, with every combination being meaningful. [...] When the features of a language are orthogonal, then the language is easier to learn and programs are easier to write because there are fewer exceptions and special cases to remember."*

IBM [12] defines orthogonality with respect to extensions on top of a base language as follows: *"An* orthogonal extension *is a feature that is added on top of a base without altering the behaviour of the existing language features. A valid program conforming to a base level will continue to compile and run correctly with such extensions. The program will still be valid, and its behaviour will remain unchanged in the presence of the*

*orthogonal extensions. Such an extension is therefore consistent with the corresponding base standard level. Invalid programs may behave differently at execution time and in the diagnostics issued by the compiler.*
*On the other hand, a* non-orthogonal extension *is one that can change the semantics of existing constructs or can introduce syntax conflicting with the base. A valid program conforming to the base is not guaranteed to compile and run correctly with the non-orthogonal extensions."*

Palsberg and Schwartzbach [23] argue that for object-oriented languages, class substitution is preferred to generic classes because it better complements inheritance as a mechanism for generating new classes:

- Just like with inheritance, class substitution can be used (repeatedly) to build new (sub-)classes (p. 147)
- Class substitution is orthogonal to inheritance (p. 147), which is made precise as follows (p. 152):
  - if a class D can be obtained from C by inheritance, then D cannot be obtained from C by class substitution; and
  - if a class D can be obtained from C by class substitution, then D cannot be obtained from C by inheritance.

The common thread in the above references is that *orthogonality* of language features embodies both *independence* and *consistent composition*. As examples we might note the following:

- *"The length of time data is kept in storage in a computer system is known as its persistence. Orthogonal persistence is the quality of a programming system that allows a programmer to treat data similarly without regard to the length of time the data is kept in storage."* [27]
- C has two kinds of built-in data structures, arrays and records (structs). It is not orthogonal for records to be able to be returned from functions, but arrays cannot. [8]
- In C, $a + b$ usually means that they are added, unless $a$ is a pointer [in which case] the value of $b$ may be changed before the addition takes place. [8]

Similar concerns are raised by Szyperski in his study of component software. He identifies an important paradigm of *independent extensibility* [29]. The essential property is that independently developed extensions can be combined (p. 84). He notes that traditional class frameworks are specialised at application construction time and thereafter disappear as no longer separable parts of the generated application. He argues for independently extensible systems to specify clearly *what* can be extended — each one of these is then referred to as a *dimension of (independent) extensibility*. These dimensions may not be orthogonal, e.g. extensions to support object serialisation will overlap extensions to support persistence.

**Orthogonality for concurrent systems** The term *orthogonality* has also been applied in other contexts, and these uses are especially pertinent to our concerns with a formalism that embodies concurrency.

In the context of the Unix shell, Raymond understands *orthogonality* to mean side-effect free [25]. In the context of extensions to the Unix C-shell, Pahl argues that *"language extension is presented as a* refinement process*. [...] The property we want to preserve during the refinement process is* behaviour*, also called* safety refinement *elsewhere. [...] Behaviour preservation is in particular important since it guarantees orthogonality of the new feature with the basic language."* [22]. In the context of state charts, *orthogonality* is presented as a form of *conceptual concurrency* which is captured as *AND-decomposition* [9].

### 3.2 Application to Petri net Extensions

It is our intention to apply the above experience from language design to the formulation of extensions to the base Petri net formalisms, so that we arrive at a set of *orthogonal* extensions. As implied by the above discussion, this is an engineering or aesthetic goal rather than a theoretical one, and its success will be measured by a number of non-functional properties, including the number of extensions that can be accommodated before the metamodel architecture needs to be refactored.

Firstly, we recall that where orthogonality is defined with respect to extensions on top of a base language it was stated that: *"An orthogonal extension is a feature that is added on top of a base without altering the behaviour of the existing language features."* In this regard, if we were considering a step semantics, we would follow the example of Christensen and Hansen who observed that for inhibitor or threshold arcs, any upper bound on a place marking ought to take into account the tokens added by the step [5]. This would be required for a step semantics if the diamond rule is to hold for concurrently enabled transitions.

Secondly, we note that orthogonality embodies both independence and consistent composition of the language elements. In our subsequent discussion, this applies to the addition of inhibitor arcs and prioritised transitions. These extensions involve disjoint attributes and therefore do not interfere with each other and can be applied in any order.

Thirdly, we note that in the context of concurrent systems and specifically the Unix shell, orthogonality has been understood as requiring extensions to be side-effect free. For this reason, we do not currently contemplate extensions like that of Reference nets as implemented in *Renew* (the Reference Nets Workshop [17]), where transitions can be annotated with arbitrary (Java) code segments which then prompts warnings in the user guide of the pitfalls of side effects.

Fourthly, in line with Pahl's approach to the Unix C-shell, we propose to capture Petri net extensions in terms of behaviour-preserving refinement. In line with earlier work [30, 18], the extension of a base net $N$ to an extended version $N'$ is defined as a morphism: $\phi : N' \rightarrow N$. A morphism respects structure and behaviour and thus the components of one object are mapped by the morphism to their counterparts in the other. It is more usual to consider the morphism as mapping the extended form to the base form. Thus, for example, the (possibly) extended or embellished set of transitions is mapped to the simpler set, rather than vice versa. Where the extension introduces new attributes and constructs then $\phi$ will essentially be a restriction mapping which ignores the additional components. Where the extension modifies existing attributes and constructs then $\phi$ will essentially be a projection mapping. Note that we do not

here consider the forms of refinement appropriate to node refinement, where a node is refined by a subnet.

For example, we might take P/T nets as our base Petri nets. If we extended these with inhibitor arcs, then, in mapping from the extended to the base form $\phi$ would ignore those arcs. On the other hand, if we extended the base Petri nets so that all tokens included a time attribute, then $\phi$ would project out that additional attribute.

In characterising Petri net extensions, we focus on the firing rule introduced above and specifically the boolean enabling function: $E : \mathcal{N} \times T \rightarrow \mathbb{B}$, where $\mathcal{N}$ is the set of Petri nets. In fact, it is more convenient to work with an extended version in terms of steps, $Y$, (i.e. sets of transitions), giving $E : \mathcal{N} \times \mathbb{P}(T) \rightarrow \mathbb{B}$.[6] We will also need to refer to the firing rule, which we characterise as a mapping from states to states: $[\ \rangle : S \rightarrow S$. Note that we use the more general term *state* in preference to *marking* because we envisage that some extensions may introduce additional state components, such as a global clock.

Our primary requirement for orthogonal extensions is that extensions maintain behavioural consistency with the base formalism:

1. $\phi(E'(N', Y')) \Rightarrow \phi(E')(\phi(N'), \phi(Y'))$
   In words: if the enabling condition holds in the extended net, then the corresponding (abstracted) enabling condition holds in the base net.
2. $S'_1[Y'\rangle S'_2 \Rightarrow \phi(S'_1)[\phi(Y')\rangle\phi(S'_2)$
   In words: if the step $Y'$ causes a change of state from $S'_1$ to $S'_2$ in the extended system, then the corresponding step $\phi(Y')$ effects the corresponding change of state from $\phi(S'_1)$ to $\phi(S'_2)$ in the base system. Note that if $\phi(Y')$ is null (because the step is part of the additional components and thus ignored by $\phi$), then this should have no effect on the base system state, i.e. $\phi(S'_1) = \phi(S'_2)$.

For simplicity, we prefer to work with a more constrained version of the first condition: $E'(N', Y') = \phi(E')(\phi(N'), \phi(Y')) \wedge E''(N', Y')$, where $E''$ supplies an *augmenting constraint* in addition to the enabling rule in the base system[7]. Of course, there is no guarantee that this approach will always be applicable, but where it is, the commutativity and associativity of the conjunction operation will facilitate the orthogonality of extensions which are disjoint (as noted above for inhibitor arcs and prioritised nets).

The essential element of the extension mechanism contemplated above is that it constitutes a form of refinement that maintains behavioural consistency, in line with Pahl[8]. He argues that *"Behaviour preservation is in particular important since it guarantees orthogonality of the new feature with the basic language"*.

We extend this to independence of multiple extensions by requiring that they can be applied in any order and the result is the same. This corresponds to requiring that

[6] While restricting our attention to an interleaving semantics, the use of steps is desirable, especially when the extension introduces additional kinds of transitions or actions (such as the elapse of time). Then the morphism will be a restriction which ignores those additional transitions and the behaviour corresponding to these transitions in the base system will be null. Thus, our steps will typically be singleton or empty sets.

[7] This has been motivated by the use of conjunction for refining class invariants and pre- and post-conditions in Eiffel [20].

[8] Quoted in section 3.1

for two morphisms, $\phi$ and $\psi$, the square of Fig. 1 commutes. As an example of this, we might consider the combination of both priorities and time to simple P/T nets. The interesting question is how the elapse of time would work in such a system — should time be allowed to elapse when no priority-enabled transition becomes disabled, or when no enabled transition becomes disabled. The former choice would contradict the requirement that we should be able to add the refinements in either order.

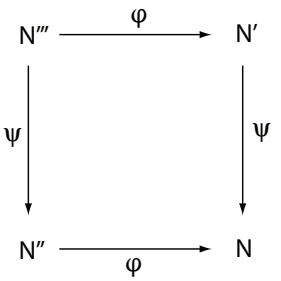



**Fig. 1.** State space for simplified Petri net for device message generation.

Against the general background of orthogonality considered in section 3.1, this approach also has the following properties:

- In line with Raymond, our extensions are side-effect free, in the sense that the only way for the extended system to affect the state of the base system is for the action of the extended system to map to an appropriate action in the base system. Thus, if a different kind of action is introduced, e.g. a procedure call, then it cannot affect the underlying marking.
- In line with IBM's definition of orthogonality, an orthogonal extension is a feature added on top of a base without altering the behaviour of the existing language features. This is especially clear if we require extensions to have an *identity* element, e.g. a prioritised net where all priorities are the same, or a timed net where the timing conditions never constrain the firing of the transition.
- The requirement by Pratt and Zelkowitz that an orthogonal language feature can be added to all other (relevant) existing constructs is *not* addressed by our proposal above — it is a matter for the language designer (or in this case, the designer of the Petri net extension). As in the case of the Unix shell, an orthogonal extension is side-effect free.

**Petri nets Firing Rule Revisited** In the context of Petri net extensions, it is our intention to ensure that extensions are side-effect free and that they will be formalised as behaviour-preserving refinements. To do so, let us revisit the definition of a firing rule in the context of the interleaving semantics [6]:

1. Computation of $T' = \cup\{t_i\}$, the set of enabled transitions. In other words $T'$ is the subset of $T$ for which $E(N \in \mathcal{N}, t_i \in T_N) = \mathit{true}$. $E$ is the enabling function

$E : \mathcal{N} \times T \to \mathbb{B}$, which has two parameters — a net within a net type $N \in \mathcal{N} = \langle P, T, Pre, Post, S \rangle$ and the transition $t \in T_N$ to which it applies. $E$ is defined as follows:

$$E(N \in \mathcal{N}, t \in T_N) : \begin{cases} True \ \ when\ t\ is\ firable \\ False\ otherwise \end{cases}$$

Note that $S$ of $N \in \mathcal{N}$ denotes the current state of net $N$. For a P/T net or a Prioritised Net, it is simply the current marking $M$, while for Time Petri Nets, it corresponds to $(M, v)$ as defined in section 2.3.
Thus firability of a given transition $t \in T_N$ can be checked. This is also the case in later definitions of enabling rules.

2. Selection of one $t \in T'$ to be fired, or some action like the elapse of time as in the case of Time Petri Nets;
3. Update of the state of $N$.

Note that Time Petri Nets allow models to evolve by firing a transition or by having time elapse. Step 2 of our firing rule caters for such alternative actions by insisting that one action be chosen at each step. Definition 12 ensures that the advance of time does not disable any already enabled transitions. In this way, we eliminate the possibility of side effects when there are two concurrent ways for the model to evolve. Thus, at this stage, the absence of side effects between enabling conditions appears to be a sufficient requirement to fit within our framework.

In the following section, we revisit in a compositional way some firing rules of well known types of Petri nets and associated features (inhibitor arcs, time and priorities management). Then, we compose them to build more elaborate Petri net types.

### 3.3 Revisiting Basic Enabling Functions and Augmenting Constraints

We now present the enabling functions for P/T nets and their augmenting constraints for inhibitor arcs, time (in the sense of [4]) and priorities (in the sense of [19]) in the framework we have developed above.

**Enabling rule for P/T nets** Let us define $E_{pt}(N, t)$ that returns true when the marking of input places is sufficient:

$$E_{pt}(N \in \mathcal{N}, t \in T_N) : \begin{cases} True \ \ iff\ \forall p \in {\bullet t}, M(p) \geqslant Pre(p, t) \\ False\ otherwise \end{cases} \tag{1}$$

**Enabling rule for priorities** Let us define the augmenting constraint $E_p(N, t)$ that returns true when $prio(t)$ has of the lowest value over the net:

$$E_p(N \in \mathcal{N}, t \in T_N) : \begin{cases} True \ \ iff\ \forall t', \rho(M, t) \leqslant \rho(M, t') \\ False\ otherwise \end{cases} \tag{2}$$

The enabling condition for P/T nets with priorities is thus:

$$E_{np}(N, t) = E_{pt}(N, t) \wedge E_p(N, t) \tag{3}$$

**Enabling rule for time conditions** Let us define the augmenting constraint $E_{tt}(N,t)$ that returns true when the local $v(t)$ associated with $t$ is in the range $[\alpha,\beta]$ (constants associated with $t$).

$$E_{tt}(N \in \mathcal{N}, t \in T_N) : \begin{cases} True \ \ iff \ v(t) \geqslant \alpha(t) \wedge v(t) \leqslant \beta(t) \\ False \ otherwise \end{cases} \tag{4}$$

The enabling condition for P/T nets with timing constraints is thus:

$$E_{nt}(N,t) = E_{pt}(N,t) \wedge E_{tt}(N,t) \tag{5}$$

Note that equation 5 is consistent with definition 11 (enabling function) assuming that the generic firing rule still allows the action corresponding to the elapse of time.

**Augmenting constraint for inhibitor arcs** Let us define the augmenting constraint $E_i(N \in \mathcal{N}, t \in T_N)$ that returns true when there are less tokens than the value specified on the inhibitor arc ($Pre_i(p,t) \neq 0$).

$$E_i(N \in \mathcal{N}, t \in T_N) : \begin{cases} True \ \ iff \ \forall p \in \bullet t \ s.t. \ Pre_i(p,t) > 0 : Pre_i(p,t) > M(p) \\ False \ otherwise \end{cases} \tag{6}$$

**Composing inhibitor arcs with defined net types** We can now combine the definition of the inhibitor arc augmenting constraint with the net types we already defined, to build P/T nets with inhibitor arcs (equation 7), Prioritised nets with inhibitor arcs (equation 8), Time nets with inhibitor arcs (equation 9), Time nets with inhibitor arcs and priorities (equation 10). More combinations can be elaborated.

$$E_{pti}(N,t) = E_{pt}(N,t) \wedge E_i(N,t) \tag{7}$$

$$E_{npi}(N,t) = E_{np}(N,t) \wedge E_i(N,t) = E_{pt}(N,t) \wedge E_p(N,t) \wedge E_i(N,t) \tag{8}$$

$$E_{nti}(N,t) = E_{nt}(N,t) \wedge E_i(N,t) = E_{pt}(N,t) \wedge E_{tt}(N,t) \wedge E_i(N,t) \tag{9}$$

$$E_{ntpi}(N,t) = E_{nt}(N,t) \wedge E_i(N,t) \wedge E_p(N,t) = E_{pt}(N,t) \wedge E_{tt}(N,t) \wedge E_i(N,t) \wedge E_p(N,t) \tag{10}$$

**Discussion** Now we have established a framework that encompasses the formal definitions of several types of Petri nets. Only interleaving semantics is considered so far. Moreover, if P/T nets constitute our base Petri nets, the framework could work with Symmetric Nets or High-Level Nets as well since the enabling functions defined in this section refer to the notion of state for which only the structure of the marking is impacted by colours. We do not detail this more due to space limitation.

In the next section, we present the MDE-based framework to compose Petri net types in a similar way to the formal framework presented here. Then section 5 shows how prioritised and time nets can be elaborated in PNML thanks to the MDE-based framework.

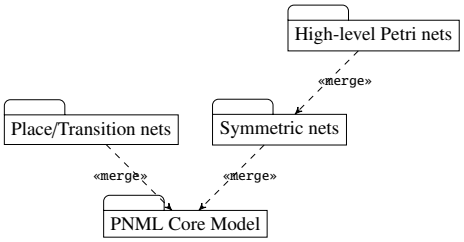



**Fig. 2.** Metamodels architecture currently defined in ISO/IEC-15909 Part 2.

## 4 A MDE Framework to Extend and Compose Petri Net Types

As highlighted in the previous section, orthogonality of augmenting constraints as well as the semantic compatibility of firing rules are crucial for the formal definitions to work properly. At the syntactic level, upward compatibility is important as well. We consider upward compatibility as the ability to extract a base net type from its extended version.

We show in this section how the formal concepts developed earlier could be projected onto the metamodel architecture of the standard, thus yielding a model-based framework to extend and compose Petri Net types.

### 4.1 Current Metamodels Architecture

Figure 2 shows an overview of the metamodels architecture currently defined in Part 2 of the standard [10]. This architecture features three main Petri net types: P/T, Symmetric and High-level Petri nets. They rely on the common foundation offered by the PNML Core Model. It provides the structural definition of all Petri nets, which consists of nodes and arcs and an abstract definition of their labels. There is no restriction on labels since the PNML Core Model is not a concrete Petri net type.

Such a modular architecture favours reuse between net types. Reuse takes two forms in the architectural pattern of the standard: `import` and `merge` package relationships, as defined in the UML 2 standard [21].

`Import` is meant to use an element from another namespace (package) without the need to fully qualify it. For example, when package `A` includes: `import B.b`, then in `A` we can directly refer to `b` without saying `B.b`. But `b` still belongs to the namespace `B`. In the ISO/IEC 15909-2 standard, Symmetric nets import sorts packages such as Finite Enumerations, Cyclic Enumerations, Booleans, etc.

`Merge` is meant to combine similar elements from the merged namespace to the merging one. For example, let us assume that `A.a`, `B.a` and `B.b` are defined. If `B` is merged into `A` (`B` being the target of the relationship), it will result in a new package `A'`:

- all elements of `B` now explicitly belong to `A'` (e.g., `A'.b`);
- `A.a` and `B.a` are merged into a single `A'.a` which combines the characteristics of both;
- actually, since `A` is the merging package (or the receiving package), `A` becomes `A'` (in the model, it is still named `A`).

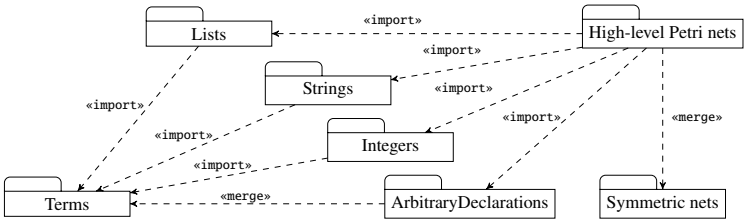



**Fig. 3.** Modular construction of High-level Petri Nets based on Symmetric Nets

`Merge` is useful for incremental definitions (extensions) of the same concept for different purposes.

In the standard, this form of reuse is implemented for instance by defining P/T nets upon the Core Model and High-level nets upon Symmetric nets, as depicted in Figure 2. Therefore, Symmetric net elements and annotations are also valid in High-Level Petri nets (but not considered as Symmetric nets namespace elements anymore).

This extensible architecture is compatible with further new net type definitions, as well as with orthogonal extensions shared by different net types. These two extension schemes are put into practice for defining Symmetric nets and High-level nets as discussed in the next section. Prioritised Petri nets, presented in sections 2.2 and 5.1 are also defined using the same extension schemes.

### 4.2 Standard Nets Types Modular Definition

With the two forms of reuse, namely *import* and *merge*, Symmetric Nets are defined in the standard upon the PNML Core Model and High-level Petri Nets are defined upon the Symmetric Nets.

Figure 3 shows how High-level Petri Nets are defined. The *merge* relationship enables the reuse of the common foundation provided by Symmetric Nets. The provided concepts are now fully part of the High-level Petri Nets namespace. Then, with the *import* relationship, sorts specific to High-level Petri Nets (Lists, Strings, Integers and Arbitrary Declarations) are integrated to build this new type.

The definition of Symmetric Nets follows the same modular approach (see Figure 4), where the package of Symmetric Nets merges the PNML Core Model and imports the allowed sorts, the carrier sets of which are finite. The `Terms` package provides the abstract syntax to build algebraic expressions denoting the net declarations, place markings, arc annotations and transition guards.

### 4.3 Extended Metamodels Architecture Framework

From the common layout of the standards metamodels, we can extract an architectural pattern which could be used to extend them in order to build new net types.

Figure 5(a) describes such a pattern. The new net type `XX Extension YY Petri Net` is built upon an existing Petri net type, which is represented by `YY Petri net` and an extension (e.g. priority, time), which is represented by `XX Extension`. The new net

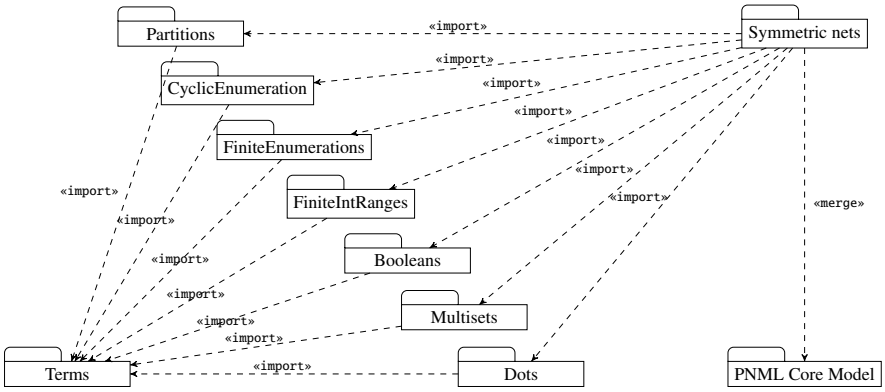



**Fig. 4.** Modular construction of Symmetric Nets based on PNML Core Model

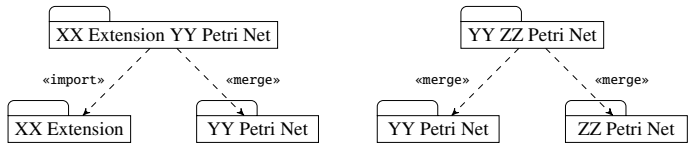
type package imports the extension package, while it merges the existing net one. This is consistent with the semantics of reuse in the standard architectural pattern, through the `import` and `merge` relationships.

Note that combining independent extensions boils down to performing a conjunct of several firing rules, which is in essence commutative. Thus, in that case, applying `XX` and `YY` extensions can be done in either order. On another hand, if one extension is further extended, then these two extensions are no longer independent, and cannot be applied in a different order.

Figure 5(b) describes the case where the new type is built upon existing net types only, which have already embedded their own extensions. The observed pattern is then composed only of the merged ones. This means all building blocks needed to build the new net type package come from the ones being merged. Therefore, no additional constructs are necessary. If any specific extension to the new net type is required, then the pattern of Figure 5(a) is applied.

## 5 The MDE-Framework applied to Nets with Priorities and Time

We now use the model-based framework previously defined to build new net types through extension and composition of metamodels. First, PT-Nets with static and dy-




(a) Based upon existing type and an extension

(b) Based only upon existing types

**Fig. 5.** Modular construction of a new net type.

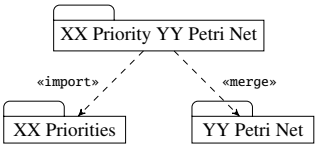



**Fig. 6.** Modular construction of prioritised Petri Nets metamodels.

namic priorities are built, then PT-Nets with time. Afterwards, we present more elaborate compositions which are the projections of some of the later equations from section 3.3.

### 5.1 Metamodels for PT-Nets with Priorities

A prioritised Petri net basically associates a priority description with an existing standardised Petri net, thus building a new Petri net type. The metamodel in Figure 6 illustrates this modular definition approach, in line with the pattern of Figure 5(a). It shows a blueprint for instantiating a concrete prioritised Petri net type, by merging a concrete Petri net type and importing a concrete priority package. The `XX Priorities` package is the virtual representation of a concrete priority package and the `YY Petri Net` is the virtual representation of a concrete Petri net type.

For example, Figure 7 shows a prioritised PT-Net using static priorities only. It is built upon a standardised PT-Net which it merges, and a `Priority Core` package, which it imports. The `Priority Core` package provides the building blocks to define `Static Priorities`, as depicted by Figure 8.

The purpose of the `Priority Core` package is to provide:

- the root metaclass for priorities, represented by the `Priority` metaclass;
- a priority level, which is an evaluated value represented as a property of the `Priority` metaclass;
- the ordering policy among the priority values of the prioritised Petri net. This ordering policy is represented by the `PrioOrderingPolicy` metaclass.

The purpose of priority levels is to provide an ordered scalar enumeration of values such that either the higher the value, the higher the priority, or the lower the value,

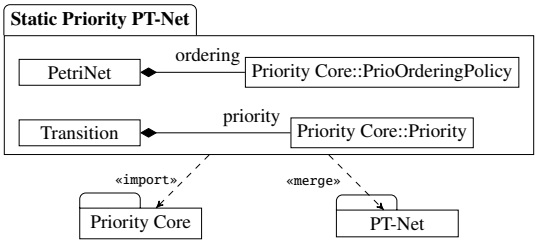



**Fig. 7.** Prioritised PT-Net metamodel showing how the priority description is attached.

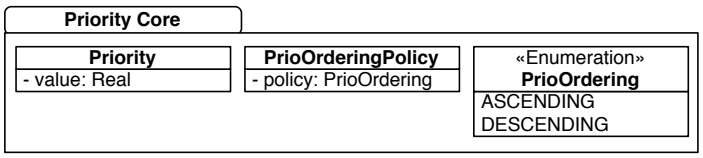



**Fig. 8.** Core package of priorities.

the higher the priority. With the `Priority Core` package, and thanks to the `Priority` metaclass, static priorities can thus be attached to transitions, as in the `Static Priority PT-Net` shown in Figure 7.

Using the same approach, Figure 9(a) shows a prioritised PT-Net which uses dynamic priorities. Dynamic priorities are built upon `Priority Core`.

This modular construction follows the extension schemes adopted so far in the PNML standard, explained earlier in this section. For instance, High-Level Petri nets build upon Symmetric nets that they merge, and new specific sorts (such as List, String and arbitrary user-defined sorts) that they import. The use of the `merge` and `import` relationships is therefore consistent.

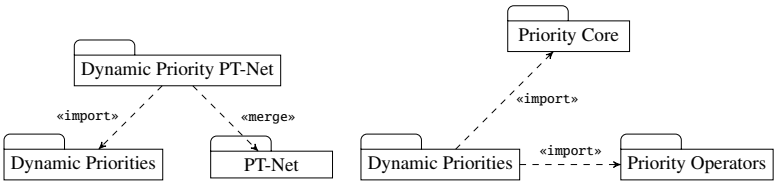
This approach is consistent with the idea that a new Petri net type subsumes the underlying one it builds upon, but the algebraic expressions it reuses are generally orthogonal to net types. Next, we introduce the metamodel for priorities.

**Priority Metamodel** Prioritised Petri nets augment other net models (e.g. P/T or Symmetric nets) by associating a priority description with the transitions. Such priority schemes are of two kinds:

- *static priorities*, where the priorities are given by constant values which are solely determined by the associated transition[9];
- *dynamic priorities*, where the priorities are functions depending both on the transition and the current net marking.

Figure 9(b) shows the modular architecture of priorities metamodels. The `Priority Core` package (detailed in Figure 8) provides the building blocks to define both `Static`

[9] For high-level nets such as Coloured nets, the priorities are given by constant values which are solely determined by the associated binding element.




(a) Prioritised PT-Net (b) Dynamic priorities

**Fig. 9.** Prioritised PT-Net metamodel using dynamic priorities

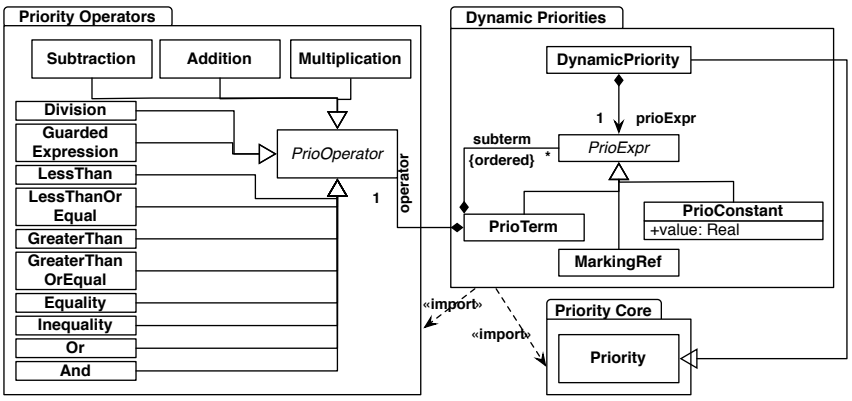



**Fig. 10.** Dynamic priorities and priorities operators packages.

`Priorities` and `Dynamic Priorities`. However, dynamic priorities are further defined using `Priority Operators`. Dynamic priorities can encompass static ones by using a constant function (for the sake of consistency in the use of priority operators).

Figure 10 shows how the `Dynamic Priorities` metamodel is built. A `Dynamic-Priority` *is a* `Priority Core::Priority`. It contains a priority expression (`Prio-Expr`). A concrete priority expression is either a `PrioTerm` which represents a term, a `PrioConstant` which holds a constant value or `MarkingRef` which will hold a reference to the marking of a place.

Note that the actual reference to the metaclass representing markings is missing. It must be added as an attribute (named `ref`) to `MarkingRef` once the concrete prioritised Petri net type is created. Its type will then be a reference to the actual underlying Petri net type `Place` metaclass, which refers to the marking.

A `PrioTerm` is composed of an operator (`PrioOperator`) and ordered subterms. This definition enables priority expressions to be encoded in abstract syntax trees (AST). For example, the conditional priority expression: `if` $M(P2) > 3$ `then` $3 \times M(P2)$ `else` $2 \times M(P1)$, is encoded by the AST of Figure 11, assuming that:

- P1 and P2 are places;
- $M(P1)$ and $M(P2)$ are respectively markings of P1 and P2.

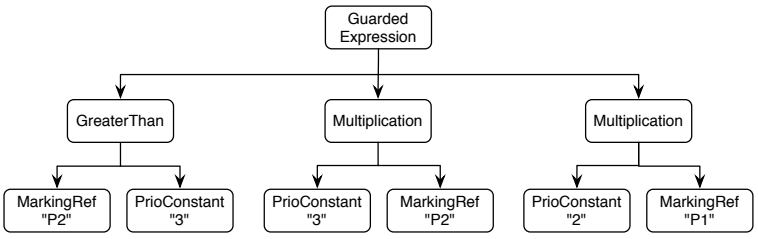



**Fig. 11.** AST of the conditional expression: `if` $M(P2) > 3$ `then` $3 * M(P2)$ `else` $2 * M(P1)$.

The priority operators are gathered within the `Priority Operators` package to allow for more flexibility in extending this priority framework. New operators can thus be added easily to this package.

Note that all these operators can also be found in ISO/IEC 15909-2, but are scattered among different sorts packages, being directly tied to the most relevant sort. For the next revision of the standard, we suggest that they be gathered in separate and dedicated packages (e.g. arithmetic operators, relational operators, etc.). This refactoring will allow for more reusability across different Petri net type algebra definitions.

### 5.2 Metamodels for PT-Nets with Time

Building a metamodel for TPNs starts with defining the metamodel for time features. We apply the same modular definition approach introduced in earlier sections. Regarding the particularly rich extension domain of TPNs, we explored several design choices. To ease the extensibility of this family of Petri nets, we sought to maximise the modularity of the definitions of the different concepts. The metamodel for time features presented in Figure 12 shows two packages, each containing a set of related features.

Package `Time4PetriNets` provides the building blocks to include time intervals and associated clocks, repectively represented by `TimeInterval` and `Clock` metaclasses. Package `Semantics4TPN` provides the different representation of the semantics for TPNs, as presented in section 2.3.

Since TPNs have a rich extension domain, `Semantics4TPN` is intended to be a flexible and easily extendable package for the different semantics. We thus have metaclasses for representing firing policies, reset policies and the server settings in TPNs. They are defined in another package which is imported by `Semantics4TPN`. That package is not shown here since it is too detailed for the granularity level of this example.

Using the metamodel for time features, we now define the metamodel for Time Petri nets, where time intervals are attached to transitions. Figure 13 shows such a metamodel, where the semantics policies are attached to the net itself, represented by the `Petrinet` metaclass. The new Petri net type package merges that for P/T nets, so as to be able to build upon existing constructs from P/T nets, within a new namespace. Note that, since a Petri net evolves using a single semantics only, it is attached to the whole model. The corresponding metaclasses are therefore attached to the net node in figure 13.

In a TPN the semantics is monoserver, strong and clocks are reset upon firing of newly enabled transitions. This can be specified by OCL constraints which will be glob-

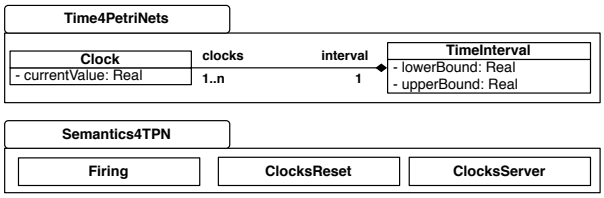



**Fig. 12.** Time features for TPNs.

ally attached to the `Semantics4TPN` package. These constraints are not presented so as to avoid cluttering the example.

These extensions of P/T nets by time features to build Time Petri nets can be removed, thus easily falling back to P/T nets, in line with the direction stated in section 3.2. With such a modular definition approach, it is easy to define the metamodel for P-time Petri nets[10] (P-TPN) and A-time Petri nets[11] (A-TPN). For instance, in the new metamodel of P-TPN, the `Place` metaclass is associated with `TimeInterval` and OCL constraints are updated so that the multiserver semantics is taken into account.

Next, orthogonality of combined features is fully implemented, through the definition of Time Prioritised Petri nets.

### 5.3 Metamodels for PT-Nets with Time and Priorities

We now consider building Dynamic Priority Time Petri nets (DPTPN), where two orthogonal extensions to P/T nets are combined. The features provided by these extensions are dynamic priorities and time. To do so, two main building blocks, already defined in earlier sections are needed:

- Dynamic Priority Petri nets, whose metamodel is shown in Figure 9(a), and
- Time Petri nets, whose metamodel is shown in Figure 13.

Figure 14 depicts the metamodel of DPTPNs, where the new Petri net type package merges the Dynamic Priority PT-Net and the TPN ones. No additional construct is needed in the new package. Every concept comes from the building blocks, i.e. the merged packages.

Indeed, in the new package, the `Transition` metaclasses from the building blocks are merged, yielding a resulting metaclass which holds a composition relationship with `Dynamic Priorities::DynamicPriority` (see Figure 10) and another composition

[10] Based on [3], time intervals are associated with places, semantics is multiserver and strong.
[11] Based on [3], time intervals are associated with arcs, semantics is multiserver and weak.

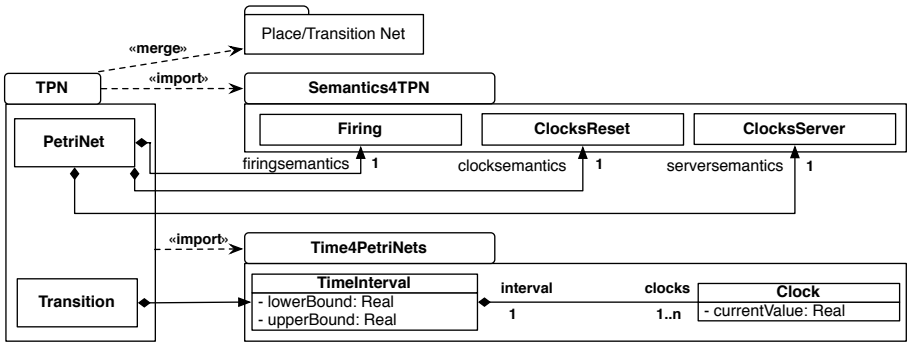



**Fig. 13.** Time Petri nets

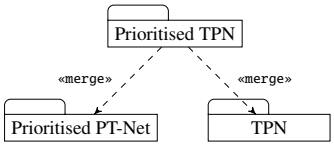



**Fig. 14.** Prioritised TPN builds upon existing orthogonal net types

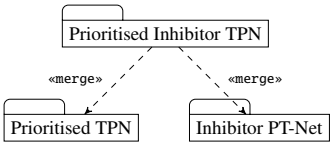



**Fig. 15.** Prioritised Inhibitor TPN builds upon existing orthogonal net types

relationship with `TPN::TimeInterval` (see Figure 13). The same applies to the resulting `PetriNet` metaclass formed by merging those from TPN and Dynamic Priority PT-Net.

Orthogonality is fully implemented in extending P/T nets through the composition of these two extensions. Whenever TPN is removed, the whole metamodel will fall back to Dynamic Priority PT-Net. Whenever Dynamic Priority PT-Net is removed, it will fall back to TPN. Whenever both extensions are removed, the metamodel will fall back to the one of P/T nets.

Next, we go one step further, by composing this new net type with special arcs.

### 5.4 Metamodel for PT-Nets with Time, Priorities and Special Arcs

Finally, we want to build DPTPNs with special arcs, such as inhibitor arcs, as defined in equation 10. But first, let us define a metamodel for inhibitor PT-Nets.

Figure 16 shows the modular construction of inhibitor PT-Net, which reflects equation 7. The `Inhibitor PT-Net` package is actually sufficiently generic to be reused, by just renaming it, to build another net type with another kind of arc (e.g. reset). It will just require importing the new extension package, as a substitute for or in addition to the `InhibitorArcExtension` one.

Every building block required to construct a metamodel for DPTPNs with inhibitor arcs is now established. Figure 15 shows the straightforward modular construction of this new net type.

## 6 Technological Issues and Discussion

This section discusses technological issues related to the implementation of this model within the Eclipse Modeling Framework [28] (EMF).

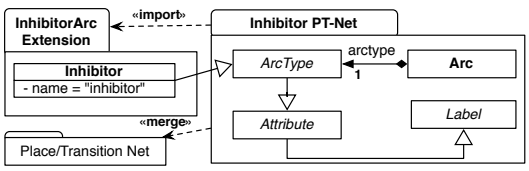



**Fig. 16.** Modular construction of inhibitor PT-Net.

**Current implementation** The approach advocated here was implemented in PNML Framework [11], which stands as one of the standard's companion tools. PNML Framework allows for handling Petri net types in a "Petri net way", in order to avoid any XML explicit manipulation. PNML Framework design and development follow model-driven engineering (MDE) principles and rely on the Eclipse Modeling Framework.

The implementation was successful, its main steps being:

1. designing the new metamodels,
2. annotating them with PNML specific information (tags),
3. assembling them and,
4. pushing a button to generate an API able to manipulate the new Petri net types.

The specific code generated by our templates to export and import models into/from a PNML file is created using the annotations decorating the metamodels in step 2. The PNML Framework plugin for Prioritised Petri nets, generated as a proof of concept for the presented extension approach, is available at http://pnml.lip6.fr/extensions.html. At this stage, it is provided as an Eclipse project with the PNML-ready source code the reader can browse and use. In the future, further updates with new net types will be provided on that web page. With the provision of our tool, it will be possible for other tool developers to define their own Petri net types, on top of existing ones.

**Technological limitations** We encountered some technological limitations during step 3 of the process because EMF does not yet offer a very convincing merge operation between models. The `EMF Compare` plug-in [7] seems to be a promising project towards this goal, but it is not yet mature enough for our use of merge. The merge had to be performed class by class, which is tedious for large metamodels.

OMG acknowledges that the UML package merge is too complex for tools to implement. Even though the Eclipse UML2 plugin currently implements this operation, it generates some inconsistencies.

Thus, to overcome this problem, we came to use the more robust `import` operation between EMF models. The procedure is the following:

1. When the composition pattern is based on merging a single base type into the new one:
   - start from the base Petri net type,
   - rename it as the new type,
   - imports the extensions.
2. When more than one base Petri net type is involved:
   - start from the one that was previously extended the most,
   - loop to step 1.

**Assessment within the standardisation process** The model driven development approach for the standard metamodels caters for extensibility maintainability, usability and reusability. However, at this stage, quantitative evaluation requires experimentations by the community. This is planned within the standardisation process, especially in part 3.

Moreover, the standardisation team aims at evaluating the opportunity to provide a more detailed description of the semantical aspects of Petri nets via the definition of the enabling functions and firing rules.

## 7 Conclusion

In this paper, we have explored extensions suitable for part 3 of the Petri Net Standard. We have proposed a framework which justifies describing these extensions as orthogonal. We have demonstrated how such extensions can be implemented in PNML Framework, an MDE-based framework which is a companion tool to the standard. This can be the stepping stone for a more general extension mechanism to integrate new Petri net types within the standard.

The experiments presented in this paper rely on Place/Transition nets as a basis for extension. We could equally well have as well chosen Symmetric Nets, or High-level Petri nets. However, the presentation would have been longer and more clumsy with no additional technical value.

Beside the immediate outcome for the Petri net community, we consider this as an interesting contribution for handling formal notations by means of Model Driven Engineering techniques. So far, metamodel management is achieved through syntactical aspects only. Our framework better captures the behavioural semantics of Petri nets by connecting the enabling rule to the attributes of the Petri net objects.

Future work aims at building a composition framework encompassing a library of existing net types and extensions, along with rules that express their semantic compatibility. This would provide safe guidelines for the construction of new net types, taking advantage of reuse, and fostering sound contributions to the standard.

**Acknowledgments** We thank Béatrice Bérard for her fruitful help with regards to the time Petri nets aspects. We would also like to thank the anonymous reviewers for their comments that enriched the paper.

## References

1. F. Bause. Analysis of Petri nets with a dynamic priority method. In Azéma, P. and Balbo, G., editors, *Proc. 18th International Conference on Application and Theory of Petri Nets, Toulouse, France, June 1997*, volume 1248 of *LNCS*, pages 215–234, Berlin, Germany, June 1997. Springer-Verlag.
2. B. Bérard, F. Cassez, S. Haddad, D. Lime, and O. Roux. Comparison of different semantics for time Petri nets. In *Automated Technology for Verification and Analysis (ATVA'05)*, volume 3707 of *LNCS*, pages 293–307, Taipei, Taiwan, Oct. 2005. Springer.
3. B. Bérard, D. Lime, and O. Roux. A Note on Petri Nets with Time. Integrated in report on WG19 plenary meeting in Paris, ISO/IEC/JTC1/SC7/WG19, 2011.
4. B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE trans. on Soft. Eng.*, 17(3):259–273, 1991.
5. S. Christensen and N. Hansen. Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs. In *14th International Conference on the Application and Theory of Petri Nets*, volume 691 of *LNCS*. Springer-Verlag, 1993.

6. I. Czaja, R. J. V. Glabbeek, and U. Goltz. Interleaving semantics and action refinement with atomic choice. In *"Advances in Petri Nets"*, pages 89–109. Springer-Verlag, 1991.
7. Eclipse Foundation, http://www.eclipse.org/emf/compare/. *The Eclipse Compare project home page*, 2011.
8. R. Green. Java Glossary: Orthogonal. http://mindprod.com/jgloss/orthogonal.html, 1996-2011.
9. D. Harel. Lecture on Executable Visual Languages for System Development, 2011. http://www.wisdom.weizmann.ac.il/~michalk/VisLang2011/.
10. L. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Trèves. A primer on the Petri Net Markup Language and ISO/IEC 15909-2. *Petri Net Newsletter*, 76:9–28, 2009. Originally presented at CPN'09.
11. L. Hillah, F. Kordon, L. Petrucci, and N. Trèves. PNML Framework: an extendable reference implementation of the Petri Net Markup Language. In *Proc. 31st Int. Conf. Application and Theory of Petri Nets and Other Models of Concurrency (PetriNets'2010), Braga, Portugal, June 2010*, volume 6128 of *Lecture Notes in Computer Science*, pages 318–327. Springer, June 2010.
12. IBM. The IBM Language Extensions, 1991. http://publib.boulder.ibm.com/infocenter/lnxpcomp/v7v91/index.jsp?topic=\%2Fcom.ibm.vacpp7l.doc\%2Flanguage\%2Fref\%2Fclrc00ibm_lang_extensions.htm.
13. ISO/IEC. Software and Systems Engineering - High-level Petri Nets, Part 1: Concepts, Definitions and Graphical Notation, International Standard ISO/IEC 15909, December 2004.
14. ISO/IEC. Software and Systems Engineering - High-level Petri Nets, Part 2: Transfer Format, International Standard ISO/IEC 15909, February 2011.
15. ISO/IEC/JTC1/SC7/WG19. *The Petri Net Markup Language home page*. http://www.pnml.org, 2011.
16. K. Jensen and L. M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer Verlag, June 2009.
17. O. Kummer, F. Wienberg, M. Duvigneau, and L. Cabac. Renew - User Guide. Technical Report Release 2.2, University of Hamburg, 2009. http://www.renew.de/.
18. C. Lakos. Composing Abstractions of Coloured Petri Nets. In M. Nielsen and D. Simpson, editors, *International Conference on the Application and Theory of Petri Nets*, volume 1825 of *LNCS*, pages 323–342, Aarhus, Denmark, 2000. Springer.
19. C. Lakos and L. Petrucci. Modular state spaces for prioritised Petri nets. In *Proc. Monterey Workshop, Redmond, WA, USA*, volume 6662 of *LNCS*, pages 136–156. Springer, Apr. 2010.
20. B. Meyer. *Eiffel: The Language*. Prentice Hall, New York, 1992.
21. OMG. *Unified Modeling Language: Superstructure - Version 2.4 - ptc/2010-11-14*, Jan. 2011.
22. C. Pahl. Modular, Behaviour Preserving Extensions of the Unix C-shell Interpreter Language. Technical Report IT-TR:1997-014, Department of Information Technology, Technical University of Denmark, 1997. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.8183.
23. J. Palsberg and M. Schwartzbach. *Object-Oriented Type Systems*. Wiley Professional Computing. Wiley, Chichester, 1994.
24. T. Pratt and M. Zelkowitz. *Programming Languages Design and Implementation*. Prentice-Hall, 3rd edition, 1999.
25. E. S. Raymond. The Art of Unix Programming, 2003. http://www.catb.org/~esr/writings/taoup/html/ch04s02.html#orthogonality.
26. P.-A. Reynier and A. Sangnier. Weak Time Petri Nets strike back! In *Proceedings of the 20th International Conference on Concurrency Theory (CONCUR'09)*, volume 5710 of *LNCS*, pages 557–571. Springer, 2009.

27. SearchStorage. Definition: Orthogonal, June 2000. http://searchstorage.techtarget.com/definition/orthogonal.
28. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Eclipse Series. Addison-Wesley Professional, $2^{nd}$ edition, Dec. 2008.
29. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
30. G. Winskel. Petri Nets, Algebras, Morphisms, and Compositionality. *Information and Computation*, 72:197–238, 1987.