Interchangeable High-Level Time Petri Nets

Christian Stehno

Parallel Systems group
Department for Computing Science
Carl von Ossietzky University Oldenburg
D-26111 Oldenburg, Germany
stehno@informatik.uni-oldenburg.de

Abstract. We present a definition for a generic Time Petri net class covering most Time Petri net types known to the community. Due to the generic approach presented in this paper, the proposed syntax may as well be used as a proposal for an expression syntax of High-level Petri nets.
Furthermore we present an implementation of translations between different Statically Timed Petri nets based on the presented syntax. This translation, implemented as XSLT schemas, provides means for interchange of net models even across Petri net data types.

1 Introduction

Petri nets have been established as a means for modelling systems in a comprehensible and easy-to-use way. Inspired by the graphical notation of Petri nets, a large number of Petri net tools have been developed. Due to the lack of a common Petri net file format every tool came up with its own format, thereby preventing exchangeability among different tools. Each tool had to provide its own set of interfaces to other file formats in order to import or export Petri nets.

The need for a universal and portable file format led to the development of the Petri Net Markup Language (PNML, [3]) which can be used to describe virtually every conveivable Petri net. Due to its generality, PNML needs a proper semantical foundation to serve as a file format which not only allows to describe Petri nets, but also to support the correct retrieval of that information from a file. PNML provides a modular extension mechanism by separating the constituent elements of Petri nets such as places, transitions, arcs, tokens, weight inscriptions, and transition guards, from the definition of Petri net classes used to describe permissible elements for some class, and the order of their occurrence.

In this paper we propose a new Petri net component to be added to PNML, namely time constraints, used in conjuction with Time Petri nets. Especially the notions of Timed High-level Petri nets give rise to the extension of time

constraints from simple number ranges to complete terms with variables and functions over the reals. All other considered time constraints can be subsumed by general mathematical expressions.

Our proposal is based on a well defined standard named MathML, briefly introduced in Sec. 1.3. MathML is an official standard by the W3C mainly used for visualizing mathematical formulas within web pages. Thus, the language is well tailored to represent arbitrary terms over the reals or subsets thereof.

The expressions, terms, and data types used for time constraints of High-level Petri nets largely correspond to those used for place types, markings, and transition guards of High-level Petri nets. In Sec. 2.3 we will hence define an XML representation for all kinds of expressions used in the domains of High-level Petri nets. Such a description is one of the current discussion topics in the ISO standardization process [12] of PNML and High-level Petri nets.

Furthermore we present an application of the time constraint extension by defining some translation algorithms on Time Petri nets. These translations transform Petri nets from one class of Time Petri nets into another one, preserving bisimulation properties of the semantics. The implementation is acting directly on the PNML representations, using a portable and flexible mechanism in terms of XML transformations.

1.1 PNML

PNML defines a flexible framework to describe arbitrary Petri nets within an XML file. In order to keep PNML as flexible as possible, the definition of PNML supports only the core elements of Petri nets. These basic elements are places, transitions, and arcs. All elements may contain an arbitrary number of labels to further specify contents and attributes of the Petri net elements.

PNML also contains a mechanism to define Petri net modules which can be instantiated to different instances, in order to efficiently describe repeated structures. Another means to apply structuring elements to nets and thus to enhance comprehensibility of the nets is the ability to split a net across different pages. Both features of PNML are not affected by the proposals in this paper and will thus not be discussed in this paper.

The extensible label mechanism of PNML allows to define arbitrary labels attached to any of the Petri net elements. All label definitions that are required to specify some class of Petri nets are collected in a Petri Net Type Definition (PNTD). Thus, PNTDs provide a formal description of PNML elements allowed or required for some PNML file to be interpreted as a net of the Petri net class described in the PNTD. This description is usually defined by an

XML Schema [10] or in some similar format. In order to ensure interchangeability of newly defined labels, a *Conventions document* keeps track of labels of common interest. Besides providing syntactic conditions for labels, semantic conditions are also defined. This ensures that labels can be interpreted in the same way in every tool. The only label defined by PNML is the `graphics` label containing optional graphical information of the Petri net element. A comprehensive description of all PNML features defined up to now can be found in [3].

1.2 Time Petri Nets

This section will not generally introduce the theory of Time Petri nets, but rather give a brief introduction of the topic and highlight the syntactic requirements of the most common classes of Time Petri nets. We will assume a basic knowledge of Petri nets in order to concentrate on the new features proposed in this paper.

A large number of different approaches to introducing time concepts to Petri nets have been proposed since the first extensions in the mid 1970s. The approaches can be roughly divided into three major classes: Statically Timed Petri nets (STPN) with constant time constraints, stochastic Petri nets (SPN) also with constant time constraints, but being evaluated using a stochastic distribution along the assigned intervals, and Dynamically Timed Petri nets (DTPN) which are defined based on High-level Petri nets [11] using values of colored tokens to define time constraints. Hybrid Petri nets [8] are orthogonal extensions of Time Petri nets and can thus be part of any of the three classes.

The time constraints of STPNs determine a time slot in which the constrained resource can be used. The most common representatives of this class are Time Transition Petri nets [15, 17] which attach time constraints to transitions. In [17], transitions are allowed to fire immediately and wait the time specified by the constraint before releasing the tokens to the postplaces. Transitions as described in [15] may not fire when they become enabled, but have to wait the specified amount of time before they can fire if not disabled in between. A similar extension has been proposed for time constraints on places [19] and on arcs [18]. In the former case tokens on constrained places become available only if they have waited the specified delay on a place, while in the latter case tokens have to wait before an arc can be used by a transition to consume such a token. All time constraints of this class are defined as an interval of the considered time domain. The actual delay can then be chosen non-deterministically from that interval.

Stochastic Petri nets [2] are plain P/T nets with black tokens and have random firing delays attached to transitions. Delays are usually defined for each transition by a rate $w \in \mathbb{R}^+$ or a weight function based on the current marking, i.e. $W : \mathbb{N}^k \rightarrow \mathbb{R}^+$. The rate is taken as the parameter of a distribution function, giving the probability of firing the transition over time. The most common distribution taken is the negative exponential resulting in e^{-wt}, but this does not hold for all SPNs. Thus, a general approach of defining time constraints for SPNs has to consider both rate and distribution as definable functions.

High-level Petri nets have been defined in order to enhance both readability and expressivity of Petri nets. In contrast to plain P/T nets, High-level Petri nets have distinguishable tokens of some data type. The most common data types are numbers, but also characters and structured types built from plain data types are possible. When firing a transition, the tokens consumed by that transition are bound to variables on the arcs. The variables can be used by the transition in order to evaluate guards. In order to fire the transition, the guard must evaluate to true. Output tokens are also determined by the guard expressions and by the variables attached to arcs leaving the firing transition. Multisets are used to allow multiple use of the same data value in an expression or when consuming tokens.

High-level nets allow the generation of compact models even for large systems, and extend the computing power of Petri nets to that of Turing machines. Both properties make High-level Petri nets attractive for system modelling, and also for industrial applications. High-level Petri nets are currently being standardized by the ISO/IEC [12].

1.3 MathML

The Mathematical Markup Language (MathML, [6]) is an XML based language allowing to describe virtually all mathematical notation and content. Due to the complexity of full MathML, only a very brief outline of the language will be provided here. The reader is referred to the official standard which also includes a thorough introduction.

Although MathML is primarily intended to serve as a backend for web presentations using HTML, MathML provides both a presentation and a content layer. The presentation layer provides support for proper mathematical typesetting. Expressions are built along their graphical layout; the semantics is frequently not recoverable from the XML description, whereas the content layer provides means to build expressions based on their mathematical syntax, and thus allows to describe expressions with unambigous semantics.

All elements of the content layer are equipped with a default visualization of the XML expressions which is however not changeable. In order to allow both, a proper semantical definition, and correct graphical layout, the two layers can be combined in one XML expression.

Most other formal languages defined for the purpose of describing mathematical expressions are proprietary file formats developed to support only one mathematical tool. Probably the only other general purpose mathematical markup language is TEX by Donald E. Knuth [14]. Although defined solely to allow visually satisfying mathematical typesetting, many of the TEX commands have been defined according to the semantics of the operators they typeset. Still, TEX is not capable of properly defining the semantics of mathematical expressions, and TEX is especially not intended to be easily extensible or comprehensible.

A mathematical markup language in addition to HTML had been proposed already in 1994, while it took some more years until the first recommendation of MathML 1.0 was published in 1998. The latest MathML version 2.0 from 2003 was enhanced by a number of new mathematical elements and clarified some questionable definitions.

The intended use of MathML in the setting of PNML is to define the semantics of mathematical expressions in different places. Thus, in the following we will concentrate on the content layer. Content elements of MathML are subdivided according to the classical mathematical domains such as algebra, calculus, and statistics. About 150 elements are defined for the content layer, which includes elements to define new functions, and links from expressions to externally defined entities.

MathML expression are written in prefix notation. The `apply` tag defines application of an operator given as the first child of the tag. All operands are subsequently listed. The use of XML attributes is limited to only a few cases, mostly to situations concerning meta notation such as definition of new identifiers. A short example in Fig. 1 gives an idea of the flexibility and expressiveness of the language. Variables, or more generally identifiers, are enclosed in `<ci>` containers while numbers use `<cn>`.

The MathML way of declaring a user defined function, and other more sophisticated MathML features, can be found in Sec. 2.

2 Petri Net Type Definitions for Time Petri Nets

Due to the enormous differences in the definitions of Time Petri nets, the possibilities of introducing timing constraints have to be defined very flexibly.

```
<apply>
  <eq/>
  <apply>
    <plus/>
    <apply>
      <power><ci>x</ci><cn>2</cn></power>
    </apply>
    <apply>
      <times><cn>4</cn><ci>x</ci></times>
    </apply>
    <cn>5</cn>
  </apply>
  <cn>0</cn>
 </apply>
```

Fig. 1. MathML code for the expression $x^2 + 4x + 5 = 0$

Although most Time Petri net models only use time constraints built from intervals over the reals, our proposal will not be limited to that kind of constraints. Moreover, it offers a flexible way to define even the interval boundaries based on arbitrary mathematical expressions.

Going through the selected Petri net types presented in Sec. 1.2 and taking into account further needs of other Time Petri net classes, we propose a definition based on a general mathematical object structure. It is thus possible to use the same syntax for expressions in various locations. Most importantly our proposal is suitable for describing expressions in High-level Petri nets. Compared to the proposal for a High-level Petri net format in [20] the MathML approach offers a better structured syntax in that all elements are proper mathematical elements. Declarations and expressions can hence be specified more flexible. A major feature of MathML is the support for user defined functions which was not handled in [20].

In the following subsections, we will present definitions for time constraints for all classes of Time Petri nets mentioned in Sec. 1.2.

2.1 STPN

For all Statically Timed Petri nets, a notion of interval is needed. MathML offers a special tag for intervals, which covers all aspects of intervals, and is superior in terms of conciseness and comprehensiveness to other solutions based on function application. The interval tag offers just the indication for the state of left and right boundaries, i.e. if one or both boundaries are open, or if they are closed, which they are by default if no state is given. The definition

of the boundaries are contained inside the tag, such that it is possible to use arbitrary expressions rather than just plain numbers. Thus, for example, it is easy to express a left-open interval from the squareroot of π to $6 * 3$ using the code given in Fig. 2. The function `root` as well as the constant `pi` are predefined in MathML. The interval definition can then be attached to either places, transitions, or arcs, depending on the Petri net model.

```
<interval closure="open-closed">
  <apply>
    <root/>
    <degree><cn>2</cn></degree>
    <pi/>
  </apply>
  <apply>
    <times/>
    <cn>6</cn>
    <cn>3</cn>
  </apply>
</interval>
```

Fig. 2. MathML code for the interval $\left]\sqrt{\pi}, 6 \times 3\right]$

2.2 SPN

Stochastic Petri nets with exponential distribution in their simplest form could be handled similarly to Statically Timed Petri nets by defining time constraints as mathematical expressions over the reals. If however more sophisticated rate functions or other distributions are considered, the use and definition of functions becomes important. There are a number of ways to integrate distribution and rate functions into PNML syntax. In order to keep the resulting PNML files short and comprehensible it would be best to declare a global function on the net level, and to refer implicitly to these functions in time constraints. Thus, only the function parameters are set by a time constraint similar to the PNML based SPN format proposed in [1]. The problem with such a proposal is that time constraints would become lists of numbers without an obvious interpretation. Most other conceivable solutions increase the redundancy of the code to some extent by adding at least the function application to the MathML code such that the parameters are not isolated anymore.

Application of predefined functions has already been shown in Fig. 2. The definition of functions in MathML is possible in different ways. An example for

a self-defined function add2 adding 2 to the parameter is presented in Fig. 3. It uses the lambda operator of MathML. The bounding variables define the parameters of the newly defined function while the last child of the tag is the defining expression.

```
<declare type="function">
  <ci>add2</ci>
  <lambda>
    <bvar><ci>x</ci></bvar>
    <apply>
      <plus/>
      <ci>x</ci>
      <cn>2</cn>
    </apply>
  </lambda>
</declare>
```

Fig. 3. MathML code for the declaration of add2(x)

2.3 High-level nets

The introduction of high level concepts does not require many syntax extensions anymore. The most obvious change in expressions compared to the ones used in time constraints for STPN and SPN is the introduction of variables. These are bound to tokens consumed by a transition upon firing. Time constraints are as well interval based, and the interval bounds are defined by expressions evaluating to numbers. Transition guards define conditions on the fireability of transitions. Guards are boolean expressions that have to evaluate to true in order to enable the transition. Except for the operators they use, numerical and boolean expressions are equivalent. All basic mathematical operators are available in MathML, thus guards and time constraints can be similarly defined in the proposed setting.

Additionally to the expressions used to express guards and time constraints the types for tokens usually attached to places have to be defined. In contrast to mathematical expressions, type definitions consist of definitions of structured sets. Several elements of MathML allow the definition of subsets constrained by boolean expressions, as in $\{x|x \in \mathbb{N} \wedge x < 5\}$ or to build sets bottom-up using set operations, as in $\{1, 2, 3\} \times (PRIMES \cup \{0\})$. The MathML code for the two sets can be found in Fig. 4.

```
<set>
  <bvar><ci>x</ci></bvar>
  <condition>
    <apply>
      <and/>
      <apply>
        <in/>
        <ci>x</ci>
        <naturalnumbers/>
      </apply>
      <apply>
        <lt/>
        <ci>x</ci>
        <cn>5</cn>
      </apply>
    </apply>
  </condition>
</set>
```

```
<apply>
  <cartesianproduct/>
  <set>
    <cn>1</cn>
    <cn>2</cn>
    <cn>3</cn>
  </set>
  <apply>
    <union/>
    <primes/>
    <set>
      <cn>0</cn>
    </set>
  </apply>
</apply>
```

Fig. 4. MathML code for set declarations

A last element of High-level Petri nets to be supported is the marking of a place. Formally, a marking of a place is defined as a multiset over the type of the place the marking belongs to. Thus, a marking of a place is a mapping from the place's type to the natural numbers giving the multiplicity for each element. A marking of a place might as well be represented as a subset of the place's type where each element may occur multiple times. MathML provides a `type` attribute as part of `<set>` which can be set to `‘‘multiset’’` in order to allow multiple occurrence of set elements. The best solution to express multiple occurrence of elements in a multiset would be an attribute for all tags to denote multiplicity as used in [20], but it does not exist in MathML. Thus, multiple occurrence of the same value has to be implemented either by multiple occurrence of the defining expressions of each value, or by application of the multiplication operator to multiset and unifying all sets.

3 Time Net Conversions

In this section, we consider implementations of translations from the different Statically Timed Petri nets into each other as laid out in [4, 7]. The net classes examined are Time Link Petri nets, Time Place Petri nets and Time Transition Petri nets. Not all conversions have been defined yet, or are even possible. The theoretical results up to now show that the class of nets with time constraints on arcs comprises both other net types, while relations between TPPN and

TTPN are not completely known [5]. The implementations shown in the next subsections thus cover only translations from TPPN and TTPN to TLPN.

The implementations presented in this paper serve different purposes. First of all, they are a case study to test the syntax proposed in the previous section. Furthermore, the implementations are meant to give an impression and an estimate of the complexity of the algorithms and their implementations.

The transformations are implemented with Extensible Stylesheet Language Transformations (XSLT, [9]). XSLT provides means for converting XML based languages into other representations, be they XML or other usually text oriented formats. A remarkable feature of XSLT is that it is itself based on XML, and thus XSLT is tailored to XML and as portable as all other XML languages. In order to introduce the main ideas of XSLT, we will give a short introduction to XSLT in the next subsection. Then, the translations from TPPN to TLPN, and from TTPN to TLPN are presented.

3.1 XSLT

Usually, XSLT translations are defined in a functional way. Therefore, XSLT transformations consist of a number of templates which specify the data they can be applied to as well as the required actions in case of a match. Input data is processed beginning with the root node of an XML tree. The best matching template is chosen. If any of the actions executed produces output using the XSLT commands, it is appended to the output document. A template may also define a set of XML nodes which are iteratively matched against the template patterns and subsequently executed. Thus, each template defines the locations it may be applied to as well as the possible successor locations data handling has to continue with.

The most recent XSLT version 2.0 from 2004 adds support for several output documents and fixes a number of extensions that had been defined before as propietary quasi-standards by various tool suppliers.

A simple template serving to copy an XML document element by element to the output document is shown in Fig. 5. The match attribute enables this template to match any XML node unless a more specific match definition is found in some other template. Then, the copy operator creates an opening tag of the matched node and all attributes are copied by the copy-of command. The select expression is expanded to all attributes (denoted by `@`) of the current node. Then all children of the current node are submitted to the match engine and subsequently processed. Finally, the closing copy tag closes the currently processed node after all children have been properly processed.

```
<xsl:template match="*">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
    <xsl:apply-templates select="child::node()"/>
  </xsl:copy>
</xsl:template>
```

Fig. 5. XSLT generic copy template

3.2 Conversion from TPPN to TLPN

TPPN can easily be converted into TLPN. The time inscriptions just need to be copied from the places to their outgoing arcs. An example is given in Fig. 6.

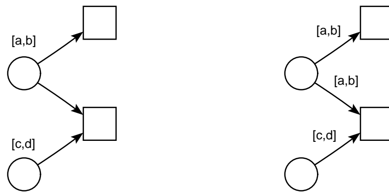

Fig. 6. Bisimular TPPN and TLPN

The implementation thus just copies the time inscriptions for each arc and omits the original ones on places. Since the number of net elements does not change, the translation is rather efficient both in terms of size and runtime.

The stylesheet consists of three templates. The standard copy template was already introduced. The two other templates are specific to the algorithm. The one shown in Fig. 7 matches all `timeConstraint` nodes and just deletes them by suppressing the application of the copy template. The more specific match expression gives priority to this template over the copy template. Since no commands are contained within the template, no output is produced.

```
<xsl:template match="timeConstraint">
</xsl:template>
```

Fig. 7. XSLT deletion template

The third template shown in Fig. 8 matches the arc nodes. It copies the node exactly as the copy template, and additionally checks if a time constraint

has to be added. The test is done by searching all place nodes for an identifier which is equal to the source identifier of the arc. Since identifiers are unique within a document, a place is only found if the arc's source is a place. In that case, the time constraint of the place is copied.

```
<xsl:template match="arc">
  <xsl:copy>
  <xsl:copy-of select="@*"/>
  <xsl:apply-templates/>
  <xsl:variable name="source" select="@source"/>
  <xsl:if test="//place[@id=$source]">
    <xsl:copy-of select="//place[@id=$source]/timeConstraint"/>
  </xsl:if>
  </xsl:copy>
</xsl:template>
```

Fig. 8. XSLT arc and time constraint copy template

3.3 Conversion from TTPN to TLPN

As shown in [4], the translation from TTPN to TLPN is possible but rather complex. The idea of the translation is to cut each place into several places which denote the presence of the token on the original place for a distinct time. The lifetime of a token is thereby spread across those new places belonging to one place in the original net. All time constraints of the preceding transitions are considered. They create an increasing sequence of points on the time scale which define the different ranges any new place will represent. A simple example is presented in Fig. 9.

Each place creates a sequence of places and transitions. The new places are created by examining all time constraints of the postset of the currently to be expanded place. All interval bounds are gathered in a numerically ordered set. Each number represents the time that may pass until the state of that specific place changes, i.e. until a transition of its postset becomes fireable or must fire. The state change is explicitly made by moving a token to the next place in the expanded net by some silent transition. A fragment of this function is presented in Fig. 10. The first part of the code collects all interval bounds from the place's postset. In order to achive this the sequence selector gathers all transitions with the correct id attribute and reads their interval bounds from the number container `cn`. The second part numerates the collected interval boundaries in ascending order. Not shown in the figure is how for each time

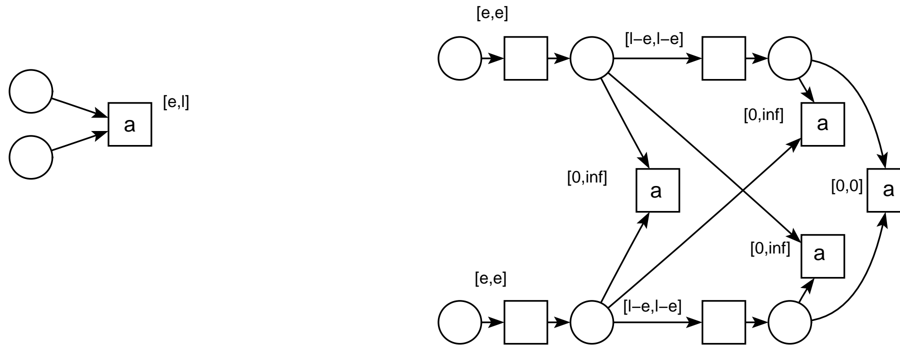

Fig. 9. Bisimular TTPN and TLPN

point a new place and, if more places will follow, a transition and two arcs are built. All newly created net elements are created in temporary documents to allow reuse in other computations. The transitions are equipped with a simple constant time constraint defined by the difference of time between the two places connected by the transition.

```
<xsl:template match="place">
  <xsl:variable name="source" select="@id"/>
  <!-- collect the existing time points -->
  <xsl:variable name="timepoints" as="xs:double*">
    <xsl:for-each select="//arc[@source=$source]">
      <xsl:variable name="target" select="@target"/>
       <xsl:sequence select="//transition[@id=$target]/timeConstraint/interval/cn"/>
    </xsl:for-each>
  </xsl:variable>

  <!-- create new places, transitions, and arcs according to time points -->
  <xsl:for-each select="$timepoints">
    <xsl:sort select="."/>
...
```

Fig. 10. XSLT fragment to get all time constraints

Transitions of the original net are duplicated such that all newly created places preserve the original firing constraints. All possible combinations of places that are representing time points of the time constraint are considered. Each of these combinations defines a new transition with the original transition's label. The time constraint for the incoming arcs of these transitions are based on the urgency state of the transition. If the place represents a point in

time which is at least the upper interval bound the transition might be urged to fire and thus gets a time constraint of $[0, 0]$. Otherwise the time constraint is $[0, \infty]$ as the transition need not fire, but is already fireable.

The algorithm is exponential in the number of places of the original net due to the fact that all possible combinations of places create new copies of the transitions. The implementation of the algorithm becomes even more complex due to the restricted linguistic elements of XSLT. Hence, XSLT variables are intensively used in conjunction with a number of features newly introduced with XSLT 2.0. It is important to note the fact that XSLT does not provide support for evaluating MathML expressions. Thus, this translation heavily relies on interval bounds given as numbers instead of arbitrary expressions in order to be able to compute with the numbers.

4 Conclusion

We have defined a sound syntax for time constraints of various types of Time Petri nets in terms of XML. The syntax is based on the widely used XML language MathML which has been defined by the W3C in order to define mathematical expressions with XML. Due to its flexibility and expressiveness, the proposed syntax can as well be used as a foundation for the definition of guard expressions and for the type definitions part of High-level Petri nets.

We presented an application of XSLT to Time Petri nets using the PNML format defined earlier. We implemented a conversion from different STPN classes into others as theoretically defined in [4, 7]. This not only shows the applicability of the proposed Time Petri net syntax, but also provides for a much wider interchangeability of PNML based Petri nets across Petri net classes.

This proposal will have to be extended to more closely refer to the forthcoming High-level Petri net standard and the existing considerations concerning High-level Petri nets in PNML. Tool support for MathML has to be further investigated, and later introduced to Petri net tools. The implementation of Time Petri net conversions has to be integrated into tools as well. One of the next releases of the PEP tool [16] will have support for XSLT, and will thus also support the Time Petri net conversions.

References

1. Mohammad Abdollahi Azgomi and Ali Movaghar. An Interchange Format for Stochastic Activity Networks Based on PNML. In Kindler [13], pages 1–10.
2. Gianfranco Balbo. Introduction to stochastic petri nets. In Ed Brinksma, Holger Hermanns, and Joost-Pieter Katoen, editors, *European Educational Forum: School on Formal Methods*

and Performance Analysis, volume 2090 of *Lecture Notes in Computer Science*, pages 84–155. Springer-Verlag, 2000.
3. Jonathan Billington, Søren Christensen, Kees van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In Eike Best and Wil van der Aalst, editors, *ATPN 03*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–506, Eindhoven, The Netherlands, 2003. Springer-Verlag.
4. M. Boyer and F. Vernadat. Language and Bisimulation Relations between Subclasses of Timed Petri Nets with Strong Timing Semantics. Technical Report 00146, LAAS, 2000.
5. Marc Boyer. Translation from timed Petri nets with intervals on transitions to intervals on places (with urgency). Technical report, Universite Denis Diderot, 2001.
6. David Carlisle, Patrick Ion, Robert Miner, and Nico Poppelier (ed.). Mathematical Markup Language (MathML) Version 2.0 (Second Edition). Recommendation, W3C, October 2003.
7. A. Cerone and A. Maggiolo-Schettini. Time-base expressivity of time Petri nets for system specification. *Theoretical Computer Science*, 216(1–2):1–53, 1999.
8. R. David and H. Alla. *Discrete, Continuous, and Hybrid Petri Nets*. Springer-Verlag, 2005.
9. Michael Kay (ed.). XSL Transformations (XSLT) Version 2.0. Working draft, W3C, November 2005.
10. David C. Fallside and Priscilla Walmsley (ed.). XML Schema Part 0: Primer Second Edition. Recommendation, W3C, October 2004.
11. Kurt Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *EATCS Monographs in Computer Science*. Springer-Verlag, 1992.
12. Ekkart Kindler. High-level Petri Nets – Transfer Format. Working draft, International Standard Organisation, 2004. ISO/IEC 15909 Part 2.
13. Ekkart Kindler, editor. *Workshop on the Definition, Implementation and Application of a Standard Interchange Format for Petri Nets*, Bologna, Italy, June 2004. Universität Paderborn.
14. Donald Ervin Knuth. *Digital Typography*. University of Chicago Press, 1999.
15. P. Merlin and D. Farber. Recoverability of Communication Protocols – Implication of a Theoretical Study. *IEEE Transactions on Software Communications*, 24:1036–1043, 1976.
16. PEP homepage. http://peptool.sourceforge.net.
17. Chander Ramchandani. *Analysis of asynchronous concurrent systems by timed nets*. PhD thesis, MIT, Boston, July 1973.
18. Joseph Sifakis. Performance Evaluation of Systems Using Nets. In Wilfried Brauer, editor, *Advanced Course: Net Theory and Applications*, volume 84 of *Lecture Notes in Computer Science*, pages 307–319. Springer-Verlag, 1975.
19. Bernd Walter. Timed Petri-Nets for Modelling and Alalyzing Protocols with Real-Time Characteristics. In *Protocol Specification, Testing, and Verification*, pages 149–159, 1983.
20. Michael Westergaard. Towards A High-Level Petri Net Type Definition. In Kindler [13], pages 71–85.