To appear in: Proceedings of the 14. Workshop Algorithmen und Werkzeuge für Petri Netze (AWPN 2007), September 2007, Universität Koblenz-Landau, Germany.

Modular PNML revisited: Some ideas for strict typing

Ekkart Kindler
Informatics and Mathematical Modelling
Technical University of Denmark
DK-2600 Lyngby
Denmark
eki@imm.dtu.dk

***Abstract*— The Petri Net Markup Language (PNML) is currently standardised by ISO/IEC JTC1/SC7 WG 19 as Part 2 of ISO/IEC 15909. But, there is not yet a mechanism for structuring large Petri nets and for constructing Petri nets from modules. To this end, modular PNML has been proposed some time ago. But, modular PNML has some problems. These problems along with ideas for their solution will be discussed in this paper.**

As a first step toward standardising a module concept for PNML in Part 3 of ISO/IEC 15909, this paper proposes a refined concept of modular PNML, which is independent of a particular kind of Petri net, but still has a strict type system. This paper focuses on the ideas and concepts; the technical details still need to be worked out. To this end, this paper also raises some issues and questions that need to be discussed before standardising modular PNML.

I. Introduction

The *Petri Net Markup Language* (*PNML*) is an interchange format for all kinds of Petri nets [5], [8], [2]. It is currently standardised as Part 2 of the International Standard ISO/IEC 15909 [4]. The *PNML Framework* helps tool makers implementing this standard [3].

However, Part 2 of ISO/IEC 15909 covers *Place/Transition nets*, *Symmetric Nets*, and *High-level Petri Nets* only. It does not cover other versions of Petri nets, such as timed or stochastic Petri nets or features such as inhibitor or reset arcs. This will be covered by Part 3 of ISO/IEC 15909. In addition, Part 3 of ISO/IEC 15909 will cover concepts for structuring large nets and for constructing nets from components.

In this paper, we discuss some ideas and concepts for defining *Petri net modules* and for constructing nets from different *instances* of such modules. This should serve as a staring point for the structuring and modularisation concepts of Petri nets that will be defined in Part 3 of ISO/IEC 15909. Actually, there was a proposal for defining and using Petri net modules quite early in the development of PNML, which was called *modular PNML* [7], [6], [8]. Since PNML is independent of a specific kind of Petri net, modular PNML was also designed to work with any kind of Petri net. The semantics of modules and the Petri nets that are constructed from instances of these modules is defined completely independent of the semantics of a particular version of Petri nets. It is defined purely syntactically by making copies of the module definitions, by connecting them via reference nodes, and by flattening the resulting net (see [7] for details) — without knowing anything about the concrete Petri net type.

The universality of modular PNML came for a price, however: The syntactical correctness of a system could only be checked after the complete system was built from the modules. Modules could be easily used in a way that was syntactically incorrect, but this would be known only after the system was complete. Therefore, modules had a very loose concept of typing. Basically, there were two reasons for this loose typing concepts: First, the focus of interfaces of modules was on places and transitions (i. e. nodes of the Petri net). Additional information attached to places or transitions, such as types, markings, or transitions, was not taken into account. Second, there was only a very simple concept for importing or exporting other information such as operators or sorts from or to a module. All this information was represented by *symbols*; but — since the module concept should be independent of a particular version of Petri nets — symbols did not have any specific structure. Therefore, the first version of modular PNML could not guarantee that symbols were combined in a syntactically correct way. Moreover, the concepts for symbols have never been worked out in full detail, since the concepts of modular PNML were not included to Part 2 of ISO/IEC 15909.

In this paper, we will discuss some ideas how the problems with the original version of modular PNML can be solved. To this end, a refined concept of symbols is introduced so that symbols can be used in a syntactically correct way—though not knowing their meaning. Then we will show how this idea carries over to labels of places and transitions. Altogether, this will result in a *strict typing* of modules: the overall syntactical correctness of a system built from modules can be checked locally in every module definition and where it is used. Still, these concepts retain one of the most important principles of PNML: it works for any kind of Petri net. For each new Petri net type, one needs to define only its *symbols* and how they can be combined. The rest of the concepts of modular PNML can be defined once and for all Petri net types.

Though these concepts are not yet worked out in full detail, they should be detailed enough for discussing the pros and cons of that approach and to discuss and decide on the future direction of the structuring and modularisation concepts that should be included to Part 3 of ISO/IEC 15909.

II. Example

Though the concepts proposed in this paper are independent of a particular kind of Petri net, we start with an example of a module using high-level Petri nets.

A. *Module definition*

Figure 1 shows an example of a module channel that transmits some information from a place *in* to a place *out*. In fact, this example is a slightly modified version of the example from [7], [8], where we represent the transmitted data explicitly now. Figure 1 shows the *definition* of the module.

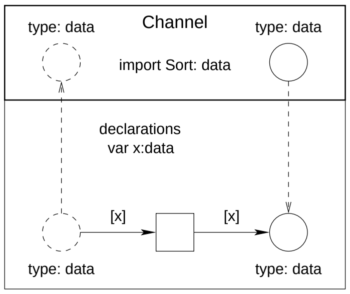

Fig. 1. The module Channel

This definition consists of two parts. The upper part in the bold-faced box defines the interface of the module and its name Channel. This interface consists of different parts: it imports a place (the dashed circle on the left-hand side) and it exports a place (the solid circle on the right-hand side). The difference between import and export nodes will become clear later in this paper. For now, it should be sufficient to know that these are the places seen and useable from outside the module: The import place will be provided by the environment of the module when it is used; the export place is a place that can be used by the environment of the module. In addition to the import and export nodes, the module definition also imports a symbol: a symbol representing a sort. As indicated by its name, this sort represents the type of data that should be transmitted over the channel. In order to make the symbol an import symbol, we use the keyword import here[1], the additional text Sort indicates that this symbol is a sort.

The lower part of the module definition in the thinly outlined box is the implementation of the module. It consists of a normal place on the right-hand side, which is the place that is exported – expressed by the dashed arrow. And there is a reference place which actually represents the imported place on the left-hand side – expressed by the dashed arrow pointing to the import place. Moreover, there is a transition between these two places; the arc annotations of the two arcs is a variable x of sort data. This variable is also defined in the implementation; in this declaration, we make use of the imported symbol for sort data.

[1]Actually, the keywords import and export and the graphical notation for import and export nodes, are not the point of this paper. In the end, the concrete syntax will be some XML and, possibly, some recommendation for the graphical representation. Since the focus of this paper is on concepts, we use some abstract syntax here.

Note that both places in the implementation of the module also have the *type* data, which exactly corresponds to the type of the import and export places in the interface. Note that we can check this net and, in particular, the labels of the places for syntactical correctness – without knowing which sort will be imported for symbol data, and which place will be imported for the import place. But, the interface requires that the imported symbol data is a sort, and the imported place has this type.

B. *Module instances*

Next, we will build a simple system from the module Channel. Figure 2 shows the use of three *instances* of module *Channel*, which are named ch1, ch2, and ch3, respectively. To indicate the instantiation, we use the name of the instance followed by the name of the module definition – a notation that is well-known from UML object diagrams. Moreover, the instances graphically resemble the interface definition of the module.

Here, we can actually understand the meaning of import places and import symbols more clearly. For each instance of the module, the import place needs to refer to some place outside the module, which will be the one imported for that instance. This is indicated by dashed arrows again. Note that export nodes of module instances are also seen from outside a module. So, we can use them for referring to them from import nodes. This way, we get a sequence of three channels. Once the data from the leftmost place are transmitted to the rightmost channel, the additional transition increments the value of that token and sends it back to the start place.

This is where the import symbol data representing a Sort comes in again. For each instance of the module Channel, we must provide a sort for the symbol data. In this example, we use the sort int, which is a built-in sort of high-level Petri nets. This way, the chain of channels transmits integer values. But, we could have used any other built-in or user-defined sort for that.

Again, we can check the syntactical correctness of this Petri net built from the module without having a look into the implementation of the modules. We need to make sure only that, for every import place, the attached type is the same as the type of the place it refers to. Since data is now bound to the sort int everywhere, this condition is obviously met.

From the model in Fig. 2 and the definition of the module Channel as shown in Fig. 1, the actual Petri net defined is the one shown in Fig. 3. It, basically, is obtained by making a copy of the module implementation for each module instance (and by prefixing all names inside the module implementation with the name of that module instance) and then merging every reference node with the node it refers to. For the nodes, this process was defined in more detail in [7], [8]. Here, we apply this idea also to symbols: Every occurrence of data in the module implementation is now replaced by the sort it is assigned in this instance of the module.

Actually, this idea was already mentioned in [7], [8]. The new idea here is that we know more about a symbol, e. g.

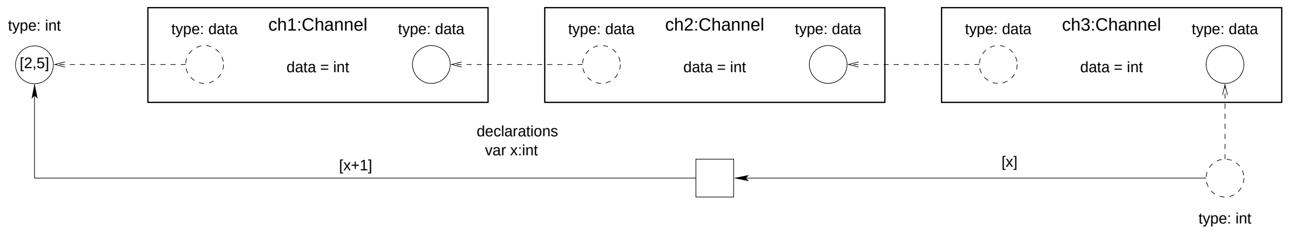

Fig. 2. Instances of module Channel

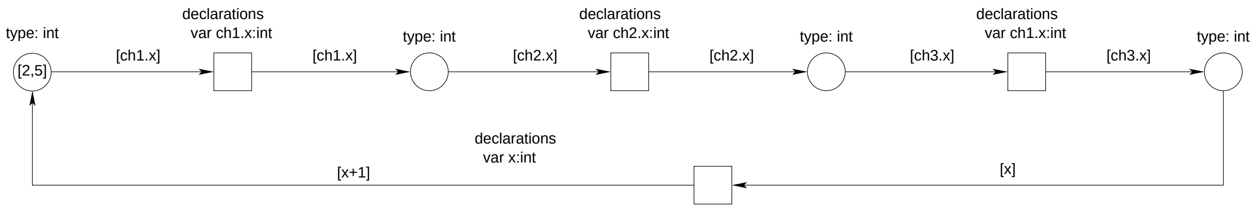

Fig. 3. The resulting model

we know that symbol data represents a sort. This way, we can make sure that module definitions and their use are syntactically correct.

In the rest of this paper, we will discuss some of the details necessary to make this idea work – independently of a specific Petri net type.

III. Concepts

In the previous section, we have seen the main concepts of modular PNML. Here we briefly rephrase the concepts again as defined in [7], [8].

A. Basic concepts

We distinguish between a *module definition* and a *module instance*. The module definition defines the *module interface* as well as the *module implementation*. The *module interface* defines *import* and *export* nodes[2]. Once a module is defined, we can use its *module instances* in other nets. We can even use module instances in the definition of other modules. The only condition on this *uses relation* among modules is that it is acyclic. This way, modules form a hierarchy.

For every import node of a *module instance*, there must be a reference to some node of the net or module which uses this module. The export nodes of the module instances can be used as if they were nodes defined in that net or module itself. Note that the users of a module do not see – or at least it is not necessary that they see – the implementation of the module. For properly using a module (syntactically), they need to know the interface definition only.

Of course, different modules might use the same identifier for naming places, transitions, or symbols. By using namespaces, the same name in different instances can be distinguished. In our example, this is indicated by prefixing all names inside a particular module instance by the name of that instance. In the original proposal of modular PNML, this concept was realised by the above mentioned prefixing mechanism, which is a bit ad hoc. Today, we could use XML namespaces for that purpose [1], but this technical issue is beyond the scope of this paper.

[2]In our example, we had import and export places only. But, in general, we can also have import and export transitions. Concerning the module concept, there is no difference between places and transitions.

B. The problem

In our example, we have used the very same idea for importing *symbols*, and we can also export symbols. The example, however, was quite simple. The imported symbol was a sort, which has no further structure. It could be used, basically, at every place where a sort was required. This is no longer true for other kinds of symbols. When we import a symbol for an operator, say f for example, it is not enough to know that it is an operator. In order to construct syntactically correct terms from this operator f, we need to know the number of parameters it takes and their types. This is but one example of a symbol with more structure.

Now, there are two question: How does modular PNML know this structure of the symbols? And, how does it know how to use a symbol in a syntactically correct way? Of course, we could built in the structure of sorts and operations to modular PNML. But, this would violate one of the principles of PNML: its independence of a specific kind of Petri nets. For example, for timed or stochastic Petri nets a module might import some delays or some firing rates for some transitions. And other types of Petri nets could have something completely different. Therefore, we cannot make specific types of symbols an integral part of modular PNML. Rather, it is necessary, to have a general concept of symbols and along with a new Petri net type, we need to define the structure of these symbols, which will be closely related to the concepts occurring in a type anyway.

In the rest, of this section, we illustrate how this can be done. We start again with a concrete example of a high-level Petri net module, where the symbols of interest are *sorts* and *operators*. Then, we show how this structure can be generalised and how the structure of the symbols can be expressed along with a Petri net type definition.

C. Structured symbols

Figure 4 shows one of my favourite Petri net examples. For some operation $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ and some value y of type B, which is put to the import place, it calculates a pair (x, y) such that $\mathsf{f}(x) = y$ and puts this pair to the output place – if such a pair exists. So, it magically computes the inverse of f for some given value y. Note that the module is independent of a particular operator f; it works for any operator, which will be provided when the module is instantiated.

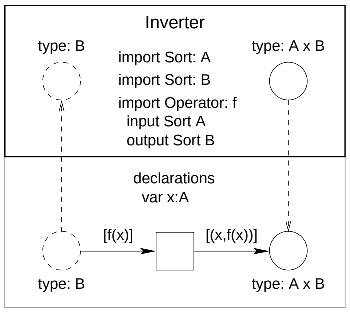

Fig. 4. The module $\mathsf{Inverter}$

Let us have a closer look at the symbols used in this example. The module definition of $\mathsf{Inverter}$ imports three symbols: two sorts A and B, and one operator f. Though, there is no specific order in which the symbols are imported, there is a dependency here. The operator f has some structure, which is made explicit in the interface. It has one input sort and one output sort. In principle, we could use any sort here. But, in this specific example, we refer to the imported sorts A and B as the input sort and output sort for that operator. Therefore, the symbol f depends on the other two symbols. The import sorts are also used for defining the type of the import place and the export place of the module. Which are sort B and the product $\mathsf{A} \times \mathsf{B}$, respectively.

The module implementation is pretty standard, except for the fact that instead of true sorts and operations, we use imported symbols. The interesting part of this example is, that we know the structure of the imported operator now; therefore, we can check the syntactical correctness of the arc inscriptions. And we can check that the sort of the arc inscriptions fit the type of the corresponding place. In order to do that, the Petri net type, here high-level Petri nets, just needs to know the input and the output sort of an operator; this is exactly the information that is provided in the module definition for the imported symbol f. From the syntactical point of view, the symbol f of high-level Petri nets is now as good as any built-in or user defined operator.

The syntax of the definition of the structure of the operator symbol f is quite verbose and appears a bit unusual. We might rather expect something like $\mathsf{import}\,\mathsf{f} : \mathsf{A} \to \mathsf{B}$. Again, this is an issue of concrete syntax, which is not the issue of this paper. Actually, an adequate concrete syntax for such definitions depends on the Petri net type and even on the particular kind of symbol. The syntax $\mathsf{import}\,\mathsf{f} : \mathsf{A} \to \mathsf{B}$ is specific to operators. The syntax used here is closer to the underlying concepts, which will become clearer in the next section, when we discuss the definition of the symbols of a particular kind of Petri net.

D. Defining symbol types

One important question is still open, however: How does modular PNML know that sorts do not have an internal structure, whereas operators have an internal structure. Even more, how does modular PNML know that an operator needs to have an input sort and an output sort; actually, we will see in a minute that there can be any number of input sorts, but there must be exactly one output sort for an operator.

In order to answer that question, we briefly revisit the way in which Petri net types can be defined in PNML[3]. A Petri net type is defined by the labels that can be attached to the different objects of the net or the net itself. The structure of these labels is defined by a UML meta model[4]. Such meta models define which labels are there in a particular version of Petri net and defines the relation between these concepts. For lack of space, we do not present the full meta model for high-level Petri nets – this alone would take over six pages. We rather have a closer look to a small fragment of it, which is shown in Fig. 5. It shows some classes of the package *Terms*, in which all concepts related to terms are defined, but some details are omitted.

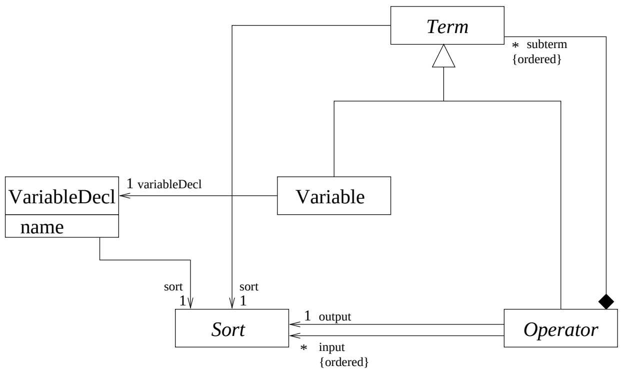

Fig. 5. Fragment from the *Terms* package

It says that a *term* can be built from a *variable* or from an *operator* and some *subterms*, where the conditions for correct typing are also omitted here. The relevant concepts, resp. classes, for our purpose are the Sort and the *Operator*, since these are the symbols we would like to import and export for high-level Petri nets. In this diagram, we can see the structure of the operator symbol immediately: the directed association

[3]Note that this Petri net type concept is used for defining three different versions of Petri nets in Part 2 of ISO/IEC 15909, but the concept for defining Petri net types itself is not standardised in Part 2 of ISO/IEC 15909. The basic concepts have been outlined in earlier papers [8] and it will be included as Part 3 of ISO/IEC 15909.

[4]Originally, the meta model was defined in terms of a RELAX/NG grammar, but now PNML uses UML meta models.

output to class Sort says that there must be exactly one output sort for an operator; and the directed association input says that there can be an arbitrary number of input sorts. Since there are no such associations for class sort, symbols for sorts do not have further structure (as far as modules are concerned). This exactly corresponds to the structure of the information that was provided for the imported operator symbol f in the example of Fig. 4.

Altogether, the information on the structure of the symbols can be derived from the meta model of the Petri net type. But, not completely: In Fig. 5, we have some other classes which we do not consider to be symbols, and, in the complete package *Terms* of the standard, there are even more. So, we need a mechanism to make explicit which concepts from the meta model should or could be used as symbols in this particular Petri net type. The simplest way is by marking the relevant classes and relevant associations by a *stereotype* ⟨⟨symbol⟩⟩ as shown in Fig. 6.

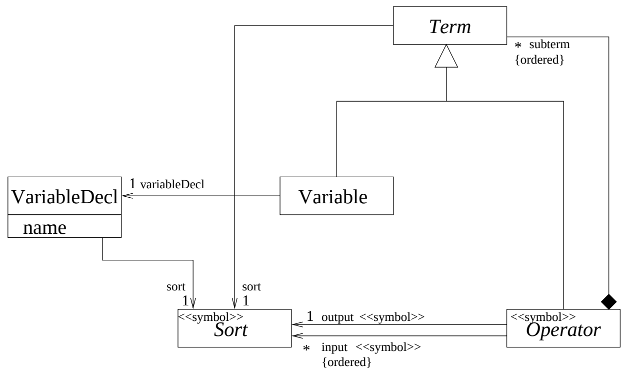

Fig. 6. Definition of symbols

Of course, there are other ways of making the symbols of a Petri net type explicit for the use with modular PNML. For example there could be two classes that inherit from Sort or Operator, respectively, and in addition inherit from some class Symbol. The latter is a more implementation oriented realisation, whereas the first one is more on the conceptual level. These details, however, are beyond the scope of this paper.

In whichever way, this concept is realised, it guarantees that symbols can be used in exactly the same way as their conceptual counterparts. This is why, syntactical correctness can be checked–without knowing its details.

IV. Issues

In the previous, section we have discussed the main idea and concepts of a module concept for PNML, which supports strict typing, i. e. syntactical correctness can be guaranteed locally without computing the full flattened model. The technical details, however, still need to be worked out. Moreover, there are some conceptual issues that need to be discussed before finalising the concepts and before implementing the technical details. In this section, we will briefly enumerate some of these issues.

Up to now, PNML ignored all labels of reference nodes and, in modular PNML, import nodes where considered to be reference nodes. Therefore, import and export nodes did not have labels or the labels did not have any meaning. Our examples, however, show that it is necessary to consider these labels and to check whether the labels of an import node fits the label of the place it refers to. The types should, basically, be the same. Therefore, it might be worthwhile to drop the idea of ignoring all labels of reference nodes and import nodes, and rather check that the labels fit to each other. The issue to be discussed here, is which labels need to be required for guaranteeing syntactical correctness and which are not. For example, names can always be ignored and they do not need to fit to each other. Types, however, should coincide. A marking might be missing; but, if present, it should be the same as in the place it refers to – sometimes we might even want to add the markings up.

In our examples, we briefly mentioned that, for one instance of a module, an import node can refer to an export node of another or even the same module instance. This, however, needs some care in order to avoid cyclic references. This can be achieved in different ways: One way would be not to allow any direct or indirect references from export to import nodes inside an implementation of a module. But, there are examples where such references make sense. If we want to have references from export nodes to import nodes inside an implementation of a module, we could make these dependencies explicit in the interface of the module. When using these modules, we could check that the references do not contain cyclic references. It needs to be discussed whether the extra effort of maintaining the dependencies between export and import nodes in the interfaces of a module is worth the effort. For deciding on this issue, we needed some convincing examples, with references from export to import nodes.

Another question is how strictly the typing should be enforced resp. mandated. On the one hand, the concepts presented in this paper could be used in such a way that, we always guarantee the syntactical correctness and strictly enforce it. But, it is not clear, whether we always want that. For example, think of a module with two imported sorts, say A and B and a place in the implementation of the module has type A, but the inscription of an outgoing arc has type B. Clearly, this is syntactically incorrect at this point. But, if someone uses this model and instantiates both import sorts A and B with the same sort, this use results in a correct overall system. Whether such modules should be allowed or not needs to be discussed.

For high-level nets and some other Petri net types with time or some similar extensions, the proposed concepts seem to have enough expressive power to construct systems in the way the are usually built. Still, it is not clear whether this is true for other kinds of Petri nets and, possibly, with completely different constructs for building systems from components. Since, modular PNML should eventually work for all Petri net types, we need to investigate some more example Petri net types in different application areas, and compare the concepts from modular PNML with other existing structuring mechanisms in Petri nets and Petri net tools.

In analogy to import and export nodes, we proposed import and export symbols for modular PNML. This could, for instance, be useful for defining some data types in some data type module that can be used in other nets without revealing the details of the data type. Still, this is more a data type issue than a Petri net issue. Therefore, this is not a too convincing example for the use of export symbols. All the other examples, we could think of were quite artificial or could be easily rephrased in terms of modules with import symbols only. So, we are not sure whether we really need export symbols at all. On the other hand, they come almost without any extra cost; therefore, there is no real argument for excluding export symbols.

At last, there are some more technical issues, which were mentioned earlier already. One is the concept of namespacing for the instances of modules. The other is the way in which concepts of a Petri net typed are distinguished or marked to be symbols in the meta model of that Petri net type. For answering these questions we need some more experience with existing XML tools and PNML implementations.

V. Conclusion

In this paper, we have revisited modular PNML. We have presented some ideas that guarantee strict typing of Petri net modules and still works for any version of Petri nets, at almost no extra cost. The only extra effort in addition to defining a new Petri net type itself is to make the concepts that can be symbols explicit in the meta model of that Petri net type.

The focus of this paper is on the ideas of modular PNML and the concepts necessary to achieve strict typing. Some details of the realisation and, in particular, the exact XML syntax for modules need to be discussed and defined.

Moreover, the issues raised in Sect. IV need a more detailed discussion and investigation, in which the Petri net community as well as the WG 19 of ISO/IEC JTC1/SC 7 should be involved. Any kind of response, suggestion, or comments are welcome.

References

[1] Namespaces in XML 1.0 (second edition). W3C recommendation, The Object Management Group, Inc., August 2006.

[2] 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 W. van der Aalst and E. Best, editors, *Application and Theory of Petri Nets 2003, 24th International Conference*, volume 2679 of *LNCS*, pages 483–505. Springer, June 2003.

[3] L. Hillah, F. Kordon, L. Petrucci, and N. Trèves. Model engineering on Petri nets for ISO/IEC 15909-2: API framework for Petri net types metamodels. *Petri Net Newsletter*, 69:22–40, October 2005.

[4] ISO/JTC1/SC7/WG19. Software and Systems Engineering - High-level Petri Nets Part 2: Transfer Format. Technical Report FCD 15909-2, v. 1.2.0, ISO/IEC, June 2007.

[5] Matthias Jüngel, Ekkart Kindler, and Michael Weber. The Petri Net Markup Language. *Petri Net Newsletter*, 59:24–29, October 2000.

[6] Ekkart Kindler and Michael Weber. Modules in pictures. *Petri Net Newsletter*, 61:5–8, October 2001.

[7] Ekkart Kindler and Michael Weber. A universal module concept for Petri nets – an implementation-oriented approach. Informatik-Bericht 150, Humboldt-Universität zu Berlin, Institut für Informatik, April 2001.

[8] Michael Weber and Ekkart Kindler. The Petri Net Markup Language. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technologies for Modeling Communication Based Systems*, volume 2472 of *LNCS*, pages 124–144. Springer, 2003.