Towards A High-Level Petri Net Type Definition

Michael Westergaard

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
Email: mw@daimi.au.dk

Abstract. The ability to exchange Petri net models between different tools has several benefits; among the most important benefits is the ability to make tools that focus on only one aspect of editing, simulation, or analysis. Another important benefit is the possibility of creating a library of common models, which can easily be used when explaining and benchmarking a new analysis method. In order to support such an exchange, a common exchange format should be agreed upon. In this paper we propose and exemplify such a format for high-level Petri nets. We also describe a number of translations from common structural constructions to the module concept of Modular Petri Net Markup Language (PNML).

Keywords: Modular PNML; High-level Petri nets; Petri net type definition; hierarchical Petri nets; fusion places; synchronous channels

1 Introduction

When it is possible to easily exchange Petri nets between different tools, it is possible to make tools focusing on only one aspect of editing, simulation, or analysis. For example, a net can be created in a tool with strong editing capabilities, simulated in a tool supporting the specific Petri net formalism, and finally be analysed using one or more strong analysis tools. Another important benefit is that it is possible to create a library of common models, which can easily be used when explaining and benchmarking a new analysis method. It is also possible to download e.g. a model implementing the TCP protocol, and use or extend it for a particular project.

In order to support such an exchange, a common exchange format should be agreed upon. PNML [1,15] provides a meta model of Petri nets and a framework to specify formats. In order to specify an exchange format, the implementer has to write a Petri Net Type-Definition (PNTD), which specifies the allowed labels for each type of element.

The PNTD for Place/Transition Petri nets (P/T net) [8] is very simple, and except for names, only specifies that places can have an initial marking and that arcs can have an arc-inscription. Both initial markings and arc-inscriptions are simply natural numbers. A PNTD for High-level Petri nets (HLPN) must not only specify more labels, such as declarations, place types, and transition guard functions, but the described labels are also more complex, as places can have arbitrary types. In this paper we consider how labels for HLPNs can be expressed.

Apart from more advanced labels, HLPNs also often allow the modeller to construct models by combining smaller components. Some such structural constructs are hierarchical nets constructed using substitution transitions and port-/socket-places [9, Chap. 3], fusion places [9, Chap. 3], and modules communicating using synchronous channels [5]. All of these constructs can be expressed by adding more labels to the involved places, transitions, or pages, but PNML already comes with a generic module concept, which can be automatically flattened to an equivalent net without modules [11]. By translating to this module concept in stead of introducing more labels, it is possible to exchange models between tools understanding elaborate composition mechanisms and tools that only understand flat nets. This paper describes how some common composition mechanisms can be expressed using the module concept of PNML.

We assume that the reader has a basic understanding of PNML and Modular PNML. The reader is also expected to have a basic intuition of high-level Petri nets.

The paper is structured as follows: In Sect. 2 we describe the labels defined for high-level Petri nets. In Sect. 3 we describe how common compositional constructs can be represented using Modular PNML, and in Sect. 4 we give our conclusions, indicate the current implementation status and some directions for future work.

2 Labels

In this section we first describe the main problems with labels in HLPNs and propose a solution. We then turn to describe the different label types required for HLPNs.

A major problem when trying to exchange labels is that different HLPN tools use different inscription languages, for example the syntax for declaring a type `INT` to be integers and two variables `n` and `m` of type `INT` in CPN-AMI [14] and CPN Tools [7, 13] can be seen in Fig. 1.

```
CLASS
  INT IS INTEGER;
VAR
  n, m IN INT;
```

(a) CPN-AMI

```
color INT = int;
var n, m: INT;
```

(b) CPN Tools

Fig. 1. Declaration of a type, `INT`, and two variables, `n` and `m`, of type `INT` in the syntax of the tools CPN-AMI and CPN Tools.

Although the concrete syntax differs, the declarations in Fig. 1 clearly express the same and can be interchanged if, instead of exchanging concrete syntax, the tools exchange abstract syntax trees (AST). The problem with similar inscription languages with different concrete syntaxes has been illustrated here using declarations, but the problem also arises for other kinds of labels.

Even though exchanging everything as ASTs solves the problem with exchanging inscriptions with different concrete syntaxes, a couple of problems arise, though. Some Petri net formalisms, e.g. Coloured Petri nets (CPN) as implemented in CPN Tools, support quite complex labels. If one want to make a simple editor that only focus on editing, parsing such labels just to be able do save nets is too much effort compared to the gains, as the translation to a simpler or just different inscription language is unlikely to be fully automatic anyway. Another problem is that adding new features is quite cumbersome because the interchange format has to be changed and all tools have to support the new format. It would be desirable if implementers could easily try out a new feature and, if it persists, the entire community could make an orderly change to the standard, rather than encourage implementers to make dozens of independent and incompatible ad-hoc "improvements". Therefore we propose allowing, at any point in the saved AST, a leaf containing verbatim text—this leaf can them be used to save concrete syntax as needed.

Apart from solving the two aforementioned problems, allowing verbatim text in the AST brings two other advantages. First, it allows for an incremental transition to the exchange format, as tool implementers can choose to save only some or parts of the labels as ASTs—even if an abstract syntax for the saved label exist (compare the declarations in Fig. 3 lines 1-11 and lines 13-20)—this may make it more difficult to move a net from one tool to another, however. Inter-operability can gradually be improved by saving more labels as ASTs. The second advantage of allowing concrete syntax everywhere is that even though the format is meant to be an exchange format for correct nets, it is also possible to use the format as primary storage format; it is highly likely that a modeller may want to save a net halfway through the modelling, but at this point some inscriptions may not be syntactically correct and therefore not parsable, so an AST cannot be constructed. If one insists on saving ASTs only, such an erroneous label cannot be saved, but it is easy to save such an inscription as verbatim text.

In Kindler's proposal for ISO/IEC 15909 part 2 [10], a cleaner cut between structured and unstructured labels is suggested: a label is either completely unstructured or completely structured. The advantage of not requiring a clear-cut separation is that it makes it possible to maintain more structure even when using an element that has not yet been standardised. For example, in Fig. 3 we are able to represent a product of integers and prime numbers structured, even though the type "prime numbers" is not standardised. If multiple tool implementors decide that the type "prime numbers" is useful, it is still easy to exchange nets between these tools—this exchange would be difficult had it not been possible to make a partially structured label. Allowing partially structured labels does not add any extra complexity to the implementation of tools, as it is very easy to write an eXtensible Stylesheet Language Transformations (XSLT) style-sheet capable of flattening structured labels to unstructured labels using a specific concrete syntax.

Now, let us turn to the description of the different label types specified for HLPNs, namely declarations, initial place markings, arc inscriptions, place types, and transition guards. Labels assigning a name to all places and transitions will not be described here. We describe all label types as a grammar only, as the concrete syntax in eXtensible Markup Language (XML) is quite easy to derive from the grammar and the examples given. We assume that a non-terminal, ⟨Unstructured⟩, produces a distinct leaf containing an arbitrary string, and that a non-terminal, ⟨Identifier⟩, produces a distinct leaf containing a legal identifier that can be used as a name. We will not go into further details about what a legal identifier is.[1]

2.1 Declarations

A declaration is an annotation belonging directly to a net or a module, and its scope is the net or module it is defined within. It is used to declare types and variables that are to be used in the net or module. As declarations can belong to a module, modules can be type-checked without any context. Using the concept of symbols from PNML, it is even possible to create parametrised modules [6].

A declaration is either a declaration of a type or a declaration of a variable, as shown in Fig. 2 lines 1-2. Definitions of functions or constants to be used in the net, as used in e.g. CPN Tools, can be saved as we allow the ⟨Unstructured⟩ production.

A declaration of a type is an identifier (the name of the defined type, e.g. `INT`) and an associated type (e.g. integer), as shown in Fig. 2 line 3. A number of standard types have been defined, among the most important are strings, integers, and subsets of integers. Types can be combined to generate e.g. products, lists, and subsets defined by an arbitrary predicate. The defined types can be seen in Fig. 2 lines 4-8.

A declaration of one or more variables is just an association between two or more identifiers, the type and the name(s) of the variable(s) that are declared to have the given type, as in Fig. 2 line 9. A variable can optionally range over multi-sets

We allow the ⟨Unstructured⟩ non-terminal in every production in order to allow implementers to maintain as much structure as possible, for example the declaration of a product of integers and the non-standard type `prime`, consisting of all prime numbers, can be defined as in Fig. 3 lines 1-11 or (less structured)

[1] A legal identifier is different for different inscription languages, and neither the union nor the intersection of the legal identifiers seem like reasonable compromises. For example, `host'send` is legal in Standard ML but not in Java, whereas `_hidden` is legal in Java but not in Standard ML. Allowing the union of legal identifiers makes it possible to save identifiers that are illegal in a concrete inscription language. On the contrary, allowing only identifiers that are legal in all inscription languages may make some naming conventions unusable, even though they are commonly used within parts of the community.

$$\begin{array}{rcl}
\langle\text{Declaration}\rangle & \longrightarrow & \langle\text{Type Declaration}\rangle \mid \langle\text{Variable Declaration}\rangle \\
 & & \mid \langle\text{Unstructured}\rangle \\
\langle\text{Type Declaration}\rangle & \longrightarrow & \langle\text{Identifier}\rangle\ \langle\text{Type}\rangle \mid \langle\text{Unstructured}\rangle \\
\langle\text{Type}\rangle & \longrightarrow & \texttt{unit}\ \langle\text{Identifier}\rangle \mid \texttt{bool} \mid \texttt{char} \mid \texttt{char interval}\ n \ldots m \\
 & & \mid \texttt{int} \mid \texttt{int interval}\ n \ldots m \mid \texttt{enum}\ \langle\text{Identifier}\rangle + \\
 & & \mid \texttt{list}\ \langle\text{Type}\rangle \mid \texttt{product}\ \langle\text{Type}\rangle\ \langle\text{Type}\rangle + \\
 & & \mid \texttt{index}\ \langle\text{Identifier}\rangle\ \texttt{interval}\ n \ldots m \\
 & & \mid \texttt{subset}\ \langle\text{Type}\rangle\ \langle\text{Predicate}\rangle \mid \langle\text{Unstructured}\rangle \\
\langle\text{Variable Declaration}\rangle & \longrightarrow & [\texttt{multiset}]\ \langle\text{Identifier}\rangle\ \langle\text{Identifier}\rangle + \mid \langle\text{Unstructured}\rangle
\end{array}$$

Fig. 2. Grammar for declarations.

as in lines 13-20. The declarations in Fig. 1 would be represented as shown in Fig. 3 lines 22-35. Lines 37-46 of Fig. 3 show how one can simulate uncoloured tokens by using the unit type to create a type that only contains one kind of tokens, in this case `e`.

2.2 Initial Place Markings and Arc Inscriptions

The initial marking for a place is an annotation of a place. It specifies the number and values of tokens on a place in the model's initial state, i.e. it evaluates to a multi-set over the elements of the place type. For example, to assign the initial marking "one token with value 2 and four tokens with value 7" to a place (of type `INT` as defined in Fig. 1) in CPN Tools or CPN-AMI, one would use the syntax from Fig. 4. Figure 5 line 1 defines initial markings, and lines 3-6 define multi-sets. A multi-set is either a value (representing one token with the given value) or a list of cardinality/value pairs.

The definition of a value is especially worth noticing. At the moment it only specifies unstructured elements, but it is designed such that future extensions are easy to make, if e.g. one wants to add types such as integer, strings, or enumerated elements. The reason that specific types have not been defined yet is that not all types are well-defined.[2] If one wants to define data-types, one should take a look at how XML Remote Procedure Call (XML-RPC) [16] or XML Schema [2] handles data-types. Another possibility is to allow more general expressions stored as an AST.

An arc inscription is an annotation of an arc, which specifies the number and values of tokens to remove from or add to a place when a transition is executed. Like an initial marking of a place, an arc expression evaluates to a multi-set over

[2] What is e.g. an integer? Java says $-2^{31} \leq n < 2^{31}$ and Standard ML says $-2^{30} \leq n < 2^{30}$

```
    <declaration>
      <type>
        <name>INTxPRIME</name>
        <type>
5         <product>
            <type><int /></type>
            <type><text>prime</prime></type>
          </product>
        </type>
10    </type>
    </declaration>

    <declaration>
      <type>
15      <name>INTxPRIME</name>
        <type>
          <text>product int * primes</text>
        </type>
      </type>
20  </declaration>

    <declaration>
      <type>
        <name>INT</name>
25      <type><int /></type>
      </type>
    </declaration>

    <declaration>
30    <variable>
        <type>INT</type>
        <name>n</name>
        <name>m</name>
      </variable>
35  </declaration>

    <declaration>
      <type>
        <name>UNCOLOURED</name>
40      <type>
          <unit>
            <text>e</text>
          </unit>
        </type>
45    </type>
    </declaration>
```

Fig. 3. The declarations of a type `INTxPRIME` as a product of integers and prime numbers (ll. 1-11 and again in 13-20), the declarations from Fig. 1 (ll. 22-35), and a type `UNCOLOURED` of unit type (ll. 37-46) in XML format.

`<2> + 4*<7>`	`1‘2 ++ 4‘7`
(a) CPN-AMI	(b) CPN Tools

Fig. 4. The multi-set "one token with value 2 and four tokens with value 7" in the syntax of the tools CPN-AMI and CPN Tools.

$$\begin{aligned}
\langle\text{Initial Marking}\rangle &\longrightarrow \langle\text{Multiset}\rangle \mid \langle\text{Unstructured}\rangle\\
\langle\text{Arc Inscription}\rangle &\longrightarrow \langle\text{Multiset}\rangle \mid \langle\text{Unstructured}\rangle\\
\langle\text{Multiset}\rangle &\longrightarrow \langle\text{Multiset Element}\rangle * \mid \langle\text{Value}\rangle \mid \langle\text{Unstructured}\rangle\\
\langle\text{Multiset Element}\rangle &\longrightarrow \langle\text{Cardinality}\rangle\, \langle\text{Value}\rangle\\
\langle\text{Value}\rangle &\longrightarrow \langle\text{Unstructured}\rangle\\
\langle\text{Cardinality}\rangle &\longrightarrow \langle\text{Non Negative Integer}\rangle \mid \langle\text{Unstructured}\rangle
\end{aligned}$$

Fig. 5. Grammar for initial place markings and arc inscriptions.

the elements of the place type. Figure 5 line 2 defines arc inscriptions in terms of multi-sets.

Various tools have different abbreviations. For example, CPN Tools allows specifying the multi-set "one token with value 5" as just `5` (as opposed to `1‘5`) and CPN-AMI allows specifying "one uncoloured token" using no arc inscription. For interchange to be possible, such abbreviations must be standardised or completely eliminated. We propose the following: empty multi-sets are saved as a multi-set node with no content (Fig. 6 lines 19-21), the multi-set "2 uncoloured tokens" is saved as the multi-set containing 2 tokens with the identifier used in the (explicit) declaration of a unit type (Fig. 6 lines 23-29). The explicit declaration used in this case can be seen in Fig. 3 lines 37-46. In other words, uncoloured tokens are not special and must be explicitly declared.

Both initial marking inscriptions and arc inscriptions can contain unstructured elements in a number of places. The recommended use is the following: if the content of an annotation is incorrect, it should be stored unstructured directly within an initial marking or arc inscription (Fig. 6 lines 1-3). If the content of an annotation is correct, but not immediately parsable as a multi-set, it should be stored unstructured within a multi-set (Fig. 6 lines 5-9). A value should be stored as a value within a multi-set (Fig. 6 lines 11-17). If the multi-set is parsable, it should be stored using multi-set elements within a multi-set, storing as much as possible structured (Fig. 6 lines 19-40). This way as much structure as possible is preserved.

2.3 Place Types

A place type is an annotation of a place, specifying the type of tokens allowed on a place. Place types can be an identifier or just a text with no further structure as shown in Fig. 7.

```
   <initialMarking>
     <text>#&*@</text>
   </initialMarking>

 5 <initialMarking>
     <multiset>
       <text>calculateInitMark();</text>
     </multiset>
   </initialMarking>
10
   <initialMarking>
     <multiset>
       <value>
         <text>2</text>
15     </value>
     </multiset>
   </initialMarking>

   <initialMarking>
20   <multiset></multiset>
   </initialMarking>

   <initialMarking>
     <multiset>
25     <value cardinality="2">
         <text>e</text>
       </value>
     </multiset>
   </initialMarking>
30
   <initialMarking>
     <multiset>
       <value cardinality="1">
         <text>2</text>
35     </value>
       <value cardinality="4">
         <text>7</text>
       </value>
     </multiset>
40 </initialMarking>
```

Fig. 6. Representations various initial markings in XML format. From the top, we see an illegal initial marking (ll. 1-3), an initial marking calculated by a function (ll. 5-9), the initial marking "one token with value 2" (ll. 11-17), the empty initial marking (ll. 19-21), the initial marking "two uncoloured tokens called e" (ll. 23-29), and the initial marking "one token with value 2 and four tokens with value 7" (ll. 31-40).

$$\langle \text{PlaceType} \rangle \longrightarrow \langle \text{Identifier} \rangle \mid \langle \text{Unstructured} \rangle$$

Fig. 7. Grammar for place types.

One could consider requiring that if a place type is specified using an identifier, a declaration of that type should exist. We have chosen not to make such a requirement, however, as most specification languages for XML do not allow the specification of such a relationship without putting some requirements on the possible values referred to, e.g. that the name is globally unique and/or satisfies some naming conventions. Furthermore it is not possible to check that the initial marking corresponds to the place type, so in any case, some checking must take place within the Petri net tool.

2.4 Transition Guards

A transition guard is an annotation of a transition, which specifies additional requirements (aside from the availability of tokens on the input arcs) for the enabling of a transition. A transition guard (defined in Fig. 8 line 1) is a, possibly empty, list of expressions (defined in Fig. 8 lines 2-7). The semantics is that all expressions of the list is combined using (possibly empty) conjunction.

$$\begin{aligned}
\langle \text{Transition Guard} \rangle &\longrightarrow \langle \text{Expression} \rangle * \mid \langle \text{Unstructured} \rangle \\
\langle \text{Expression} \rangle &\longrightarrow \langle \text{Expression} \rangle \, \texttt{and} \, \langle \text{Expression} \rangle \\
&\quad \mid \langle \text{Expression} \rangle \, \texttt{or} \, \langle \text{Expression} \rangle \\
&\quad \mid \texttt{not} \, \langle \text{Expression} \rangle \\
&\quad \mid \langle \text{Expression} \rangle \, \texttt{=} \, \langle \text{Expression} \rangle \\
&\quad \mid \texttt{add} \, \langle \text{Expression} \rangle \, \langle \text{Expression} \rangle \\
&\quad \mid \langle \text{Value} \rangle \mid \texttt{true} \mid \texttt{false} \\
&\quad \mid \langle \text{Unstructured} \rangle
\end{aligned}$$

Fig. 8. Grammar for transition guards.

An example guard expressing $m = n \vee (m > 0 \wedge n > 0)$ and $p > 0$ (a list of 2 expressions) is shown in Fig. 9.

We note that expressions are not type-safe, and only a small part of the desirable functions are specified. For example, in Fig. 9 lines 17, 20, and 27, we had to write the function $>$ using (XML escaped) concrete syntax. For future

```
<guard>
  <expression>
    <or>
      <expression>
        <equals>
          <expression>
            <text>m</text>
          </expression>
          <expression>
            <text>n</text>
          </expression>
        </equals>
      </expression>
      <expression>
        <and>
          <expression>
            <text>m&gt;0</text>
          </expression>
          <expression>
            <text>n&gt;0</text>
          </expression>
        <and>
      </expression>
    </or>
  </expression>
  <expression>
    <text>p&gt;0</text>
  </expression>
</guard>
```

Fig. 9. How the expression $m = n \vee (m > 0 \wedge n > 0)$ and $p > 0$ is represented using XML (in XML, "$>$" is represented as "&gt;").

work, an important part is specifying more expressions and splitting them up in parts suitable for different formalisms.

2.5 Inhibitor Arcs, FIFO Places, Prioritised Transitions, etc.

A lot of different kinds of arcs, places, and transitions are defined [3, 4, 12]. We do not believe that these are unique to HLPNs, and they shall therefore not be treated in detail here.

Most of these can quite easily be specified using attributes defined in a shared conventions document, as suggested in [1]. For some of the special objects, e.g. prioritised transitions, additional annotations may be required (the priority), but this is not high-level Petri net specific, and therefore not treated further here.

3 Structural Constructs

Aside from using elaborate types and arc expressions to fold nets, an important feature of High-level Petri nets is the ability to compose smaller modules into the complete net, possibly using some of the components multiple times. Composition is traditionally obtained in different ways, the most important being composition using substitution transitions and port/socket assignments (hierarchical Petri nets), composition using fusion (shared) places, and communication using synchronous channels (shared transitions).

While all the above constructs easily can be made by introducing a number of labels, we find that it is important to avoid this and use the module concept of PNML instead. By using the module concept of PNML, we make it possible to share nets between tools supporting very elaborate structural constructs and tools that only support flat nets, as it is possible to mechanically flatten a Modular PNML net to a Basic PNML net, i.e. a net without any modular constructs as described in [11].

In this section we describe how hierarchical nets and fusion places can be realised using PNML. We also describe some of the problems with realising synchronous channels as described in [5] using the current version of PNML and suggest two methods for overcoming this problem.

The constructions used here are not unique to high-level nets, and can easily be used to also introduce advanced composition techniques in other net types, e.g. P/T nets, without breaking compatibility with tools without the features thanks to the automatic flattening.

3.1 Hierarchical Petri Nets

By using hierarchical Petri nets, a modeller is able to make multiple instances of a module. The idea is that we create a sub-page (e.g. modelling a host) with an interface. The interface is created by making some of the places on the sub-page into so-called ports. We can then create a net by creating a page (e.g. with some network structure) containing one or more substitution transitions. A number of places on this super-page, called sockets, is then assigned to the ports on the sub-pages represented by the substitution transitions. A good example is the Ring Network example from [9, Chap. 3]. Informally, the semantics of hierarchical Petri nets is that we replace the substitution transition by copying the contents of the sub-page it represents. Each port/socket pair is converted into a single place, whose set of arcs is the union of the arcs of the port and the socket.

This module construct resembles the module construct of PNML, which makes a translation quite simple. We simply create a module for each sub-page (*not* one for each instance of the module!), with an interface consisting of the port places. The substitution transition is converted to an instance, and the port/socked assignments is a parameter assignment.

The only thing we need to consider is whether the port places should be represented as import or export places. Import places allow modules to import a place from the surroundings as part of the module, whereas export places exhibit parts of the module to the surroundings. We have chosen to use import places, because we can then have different labels for different instances of the module, in particular different initial markings, as the import place simply refers to a place on the instantiating page. Using export places would allow us to have some ports on the sub-page not assigned to a socket on the super page, which is not possible using import places, as the import place would be a dangling pointer. Export places does not allow us to have different initial markings for different instances as the real place resides on the sub-page. We can simulate unassigned ports when using import places by automatically creating a place on the super-page, which is assigned to the port.

3.2 Fusion Places

Fusion places or shared places have multiple participants, but only one actual instance. Such a set of participants is called a fusion set. Several semantics for fusion places exist, differing in the scope of the sharing. Here we will only deal with global fusion places, i.e. where all participants belonging to a fusion set share a single place instance. Other kinds of fusions share a single instance between participants on a single page, either globally or only per instance. Using fusion places, it is possible for two or more modules to communicate by adding/removing tokens to/from shared places.

The intuition of our construction is to construct a page for each fusion set, containing only a single global place. We then translate each participant into a reference to the single global place. Global nodes can be referred without explicitly passing a reference, nicely modelling how we think of fusion places: a single real place, and a number of participants referring to it.

This approach has several advantages. When loading a net with fusion places, even in a tool that does not support fusion sets nor Modular PNML, adding and removal of participants to the group is quite easy. Adding a participant amounts to just making a reference (to the place representing the fusion set), whereas removal amounts to removing the reference. This resembles how one would do in an object oriented programming language: we would just remove the reference to the fusion set and have the garbage collector remove the fusion set when no other participant refers to it.

One might think of other ways to model fusion places if we want to avoid the overhead of a new page. For example we could make one participant canonical by converting it to a real place and have the other participants refer to it. This introduces a lot of new overhead though. For example, when the canonical place is removed, we need to make another participant canonical and globally update the references. This also makes removal of the last participant from a fusion set cumbersome, as we have to detect that no other participants exist. Creating a new place and have all others refer to that would solve these problems, but would give problems if the page it resides on is deleted, which puts us back to having to create a special page as described above.

3.3 Synchronous Channels

Synchronous channels can be thought of as transition fusion, but the scope is not, as for fusion places, global. A synchronous channel corresponds to a fusion set of transitions, but the participants are separated into two groups, namely `!?` and `?!`, which can be thought of as sender and receiver. The semantics is that each sender/receiver pair shares an instance of a transition, meaning each sender, independently of the other senders, can send to any receiver. See e.g. [5] for a producer/consumer example behaving like this. For a producer/consumer system with 2 producers and 3 consumers, we would then have $2 \cdot 3 = 6$ actual instances.

The immediate solution is to handle synchronous channels just like fusion places, meaning we would create a page containing a single transition, and let all the participants refer to this. This approach does not work, however, as we then require all participants to be fully synchronised; in our producer/consumer example above, it would require that all producers were willing to produce, and all consumers ready to consume for a single transfer to happen, which is clearly not the intended semantics.

Creating a global node for each sender/receiver pair is not quite enough either, as we might need to refer to more than one global node (in the producer/consumer example, each producer would need to refer to 3 global transitions, one for each consumer).

This suggests we might need a more elaborate way to make global nodes. We need to be able to make a reference node that is able to refer to more than one node, by simply copying the reference and all arcs for each reference. This can be formalised, e.g. by writing an XSLT style-sheet. Formalising it like this has the obvious advantage that tools that do not support the extension can easily be given a flattened version.

4 Conclusion

In this paper we have introduced a PNTD for high-level Petri nets. We have treated the problems that arise when trying to share inscriptions between tools with different but similar inscription languages. Saving abstract syntax only enables sharing inscriptions between such tools, but makes it difficult to try new features. We have therefore argued that an approach where we save inscriptions in abstract syntax, but allows the escape to concrete syntax when needed. We have introduced the labels required for high-level Petri nets, shown a specification and examples of use.

Furthermore we have given a small catalog of translations from common structural constructs of high-level Petri nets to the module concept of Modular PNML, namely how substitution transitions, fusion places, and synchronous channels can be represented.

A nearly complete PNTD for HLPNs has been implemented, and further work is currently happening in two different settings. First, an XSLT style-sheet, translating the current format of CPN Tools to the proposed format has been developed. Second, the format is seriously considered for a future major release of CPN Tools. The PNTD, a style-sheet converting a net from CPN Tools into the proposed format, a style-sheet flattening structured labels to the concrete syntax of CPN Tools, and a number of examples can be obtained from `http://www.daimi.au.dk/~mw/local/pnml/`.

The XSLT style-sheet translating from CPN Tools has been used as a basis for defining two of the described structural constructs, namely fusion places and hierarchical nets. It is expected to be integrated into the converter shipping with CPN Tools, thereby allowing almost immediate export of high-level nets generated by CPN Tools.

The translation is also used in a project to add new composition mechanisms to CPNs by transformations on Petri nets saved in the presented format.

Considering the labels defined in Sect. 2, two areas need improvement, namely values and expressions. The problem can be simplified by allowing arbitrary expressions where only values are allowed currently. Then we need to work on extending expressions, starting by extending with the usual arithmetical operators (e.g. addition, subtraction, multiplication, division, modulo, and increment), some more boolean operators (e.g. exclusive or), and possibly some string operators (e.g. concatenation), list operations and other more advanced operations. We also need to consider how expressions can be divided into reasonable chunks, suitable for well-formed nets and other kinds of nets supporting more elaborate expressions. This is perhaps one of the most important parts of future work, but can easily be done incrementally, because of the ability to escape to concrete syntax at any time.

It would also be nice to be able to type-check expressions. In theory this is possible using most validation languages, but implementing an external type checker might be much easier.

Apart from the structural constructs described in Sect. 3, it would be nice to describe other constructs as well, e.g. substitution places or constructs not commonly used in Petri nets, such as iteration of nets.

Acknowledgements. The author wishes to thank the members of the CPN Group in Aarhus and the participants of various standardisation meetings for valuable discussions about an interchange format for HLPNs. Also thanks to the reviewers for valuable comments on an earlier version of this paper.

References

1. J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In *Proc. of ICATPN 2003*, volume 2679 of *LNCS*, pages 483–505. Springer-Verlag, 2003.
2. P.V Biron and A. Malhotra. XML Schema Part 2: Datatypes. `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/`.
3. G. Chiola, S. Donatelli, and G. Franceschinis. Priorities, Inhibitor Arcs and Concurrency in P/T nets. In *Proc. of ICATPN 1991*, pages 182–205, 1991.

4. S. Christensen and N.D. Hansen. Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs. In *Proc. of ICATPN 1993*, volume 691 of *LNCS*, pages 186–205. Springer-Verlag, 1993.
5. S. Christensen and N.D. Hansen. Coloured Petri Nets Extended with Channels for Synchronous Communication. In *Proc. of ICATPN 1994*, volume 815 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 1994.
6. S. Christensen and K.H. Mortensen. Parametrisation of Coloured Petri Nets. DAIMI-PB 521, Department of Computer Science, University of Aarhus, 1997.
7. University of Aarhus CPN group. CPN Tools homepage. `http://www.daimi.au.dk/CPNTools/`.
8. J. Desel and W. Reisig. Place/Transition Petri Nets. In *Lecture on Petri nets I: basic models*, volume 1491 of *LNCS*, pages 122–173. Springer-Verlag, 1998.
9. K. Jensen. *Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. Volume 1: Basic Concepts.* Springer-Verlag, 1992.
10. E. Kindler. High-level Petri Nets—Transfer Syntax. Proposal for the International Standard ISO/IEC 15909 Part 2. Draft Version 0.3.0. `http://wwwcs.upb.de/cs/kindler/publications/liste.html#04`, April 2004.
11. E. Kindler and M. Weber. A universal module concept for Petri nets. An implementation-oriented approach. *Informatik-Berichte*, (150), June 2001.
12. M.A. Marsan, G. Chiola, and A. Fumagalli. The semantics of capacities in P/T nets. In *Advances in Petri Nets 1989, covers the 9th European Workshop on Applications and Theory in Petri Nets, held in Venice, Italy in June 1988, selected papers*, volume 424 of *LNCS*. Springer, 1990.
13. A.V. Ratzer, L. Wells, H.M. Lassen, M. Laursen, J.F. Qvortrup, M.S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In *Proc. of ICATPN 2003*, volume 2679 of *LNCS*, pages 450–462. Springer-Verlag, 2003.
14. The MARS Team. CPN-AMI Home Page. `http://www-src.lip6.fr/logiciels/mars/CPNAMI/`.
15. M. Weber. Petri Net Markup Language. `http://www.informatik.hu-berlin.de/top/pnml/`.
16. D. Winer. XML-RPC Specification. `http://xmlrpc.org/spec`.