The Petri Net Markup Language

Michael Weber[1,⋆] and Ekkart Kindler[2]

[1] Humboldt-Universität zu Berlin, Institut für Informatik
mweber@informatik.hu-berlin.de
[2] Technische Universität München, Fakultät für Informatik
kindler@informatik.hu-berlin.de

Abstract. The *Petri Net Markup Language (PNML)* is an XML-based interchange format for Petri nets. PNML supports any version of Petri net since new Petri net types can be defined by so-called *Petri Net Type Definitions (PNTD)*.
In this paper, we present the syntax and the semantics of PNML as well as the principles underlying its design. Moreover, we present an extension called *modular PNML*, which is a type independent module concept for Petri nets.

1 Introduction

One of the most required features of Petri net tools are functions for exporting Petri nets to other tools and for importing nets from other tools. The problem with this apparently simple and purely technical feature is the multitude of different *Petri net types* and the multitude of different tools and file formats for these different net types. This makes it impossible to provide all desired import and export functions (with reasonable expenses). This situation was the starting point of a standardization effort launched during the International Conference on Application and Theory of Petri Nets 2000 with several proposals for XML-based interchange formats [1].

The *Petri Net Markup Language (PNML)* was one of these proposals, which focussed on the problem of the different Petri net types. Since that time, we have worked out the details and have implemented PNML as the file format for the Petri Net Kernel [5,10]. In this paper, we discuss the concepts of PNML and present its syntax and semantics.

The design of PNML was governed by the following principles:

Readability. The format should be human readable and editable with a conventional text editor.

Universality. The format should not exclude any version of Petri nets. Rather, it should be possible to represent any version of Petri nets with any kind of extensions.

⋆ Supported by the Deutsche Forschungsgemeinschaft within the project 'Petri Net Technology'.

Mutuality. The format should allow us to extract as much information as possible from a Petri net – even if the Petri net type is unknown. Therefore, the format must extract the common principles and the common notations of Petri nets.

Clearly, the use of XML guarantees the readability of the format[1]. Universality can be guaranteed by attaching all additional information of a particular Petri net type to the *objects* of the net. This is achieved by *labelling* net objects and the net itself. The legal labels, their possible values, and the possible combination of values are defined by a *Petri Net Type Definition (PNTD)*. Mutuality can be guaranteed by *conventions*, which are a set of standardized labels. Technically, the conventions are an extensible collection of possible labels along with a description of their semantics and their typical use. Then, a new PNTD can be built from these labels and, possibly, some new ones.

Another important issue for interchange formats is the size of real world systems. Typically, real world systems are too large to be drawn on a single page – even if the page is unbounded in principle. Therefore, tools have mechanisms for editing large systems. But, each tool provides a different mechanism. To cope with this problem, PNML provides a net type independent mechanism for editing and structuring large nets. Actually, there are two mechanisms, a simple one and a more flexible one:

Pages and references. *Pages* and *references* allow the user to draw a net on different pages and to relate these nets by merging some nodes via so-called *references*. Due to its simplicity and because most tools support similar mechanisms, it should be easy to export and import those nets by any tool. Therefore, this mechanism is part of PNML. The problem with this simple concept, however, is that it does not support abstraction. It basically allows us to draw large nets without any structure.

Modules. In many cases, the use of *modules* is more convenient because the same module can be used several times, once defined. Thus, a system can be built recursively from *module instances*, which reflects the way engineers build systems. Therefore, modules support abstraction much better than the simple page concept. Since the module concept is more evolved, it might not be supported by most tools. But, we provide a semantics in terms of pages and references, which could be run as a preprocessor for those tools not supporting *modular PNML*.

Pages and references[2] are a widely used concept for drawing large systems. Many of today's tools support this concept with a similar semantics. Maybe, the use of pages has become popular by Design/CPN [4]. Modules, however, are not widely used, and there is no unique semantics for modules. The reason is that, in

[1] To be honest, the true argument for using XML is its popularity, which is a good sales argument.

[2] References are sometimes called merge nodes.

most cases, the module concept exploits the special features of a particular Petri net type. In particular, this applies to module concepts that support dynamic creation of module instances at runtime such as in object nets of Renew [11] or in higher-order nets of the Moses project [9]. Here, we present a module concept that works for any Petri net type. Therefore, we restrict ourselves to static instances (module instances are created at the buildtime of a system only). This concept is similar to the module concept used for Signal/Event-systems [7]; it is more general, however, because it supports modules with parameters and because it is not restricted to a particular Petri net type.

2 Concepts

The concepts of PNML are independent from its syntactic representation. In particular, it is independent from XML. Therefore, we discuss its concepts and its terminology, first. Its XML syntax will be discussed in Sect. 3.

Remember that universality is one of the main principles of PNML. Therefore, PNML must be sufficiently general to represent all versions of Petri nets, on the one hand. On the other hand, mutuality requires to capture the essence of Petri nets and to exclude all kinds of nonsense, which does not represent a Petri net at all. This is achieved by providing a general format, which is restricted to the specific needs of a particular version of Petri net by defining a Petri net type.

2.1 General format

We start with the general format. Basically, the general format of PNML is a labelled graph with two kinds of nodes: places and transitions. But, there are many more concepts, which will be explained in the following paragraphs. Figure 1 gives an overview on all concepts, which can serve as a road-map while reading the following paragraphs.

Petri nets and objects. A file that meets the requirements of the interchange format is called a *Petri net file*; it may contain several *Petri nets*. Each Petri net consists of *objects*, where the objects, basically, represent the graph structure of the Petri net. Thus, an object is a *place*, a *transition*, or an *arc*. For structuring a Petri net, there are three other kinds of objects, which will be explained later in this section: *pages*, *reference places*, and *reference transitions*. Each object within a Petri net file has a unique *identifier*, which can be used to refer to this object.

For convenience, we call a place, a transition, a reference place, or a reference transition a *node*, and we call a reference place or a reference transition a *reference node*.

Labels. In order to assign further meaning to an object, each object may have some *labels*. Typically, a label represents the name of a node, the initial marking

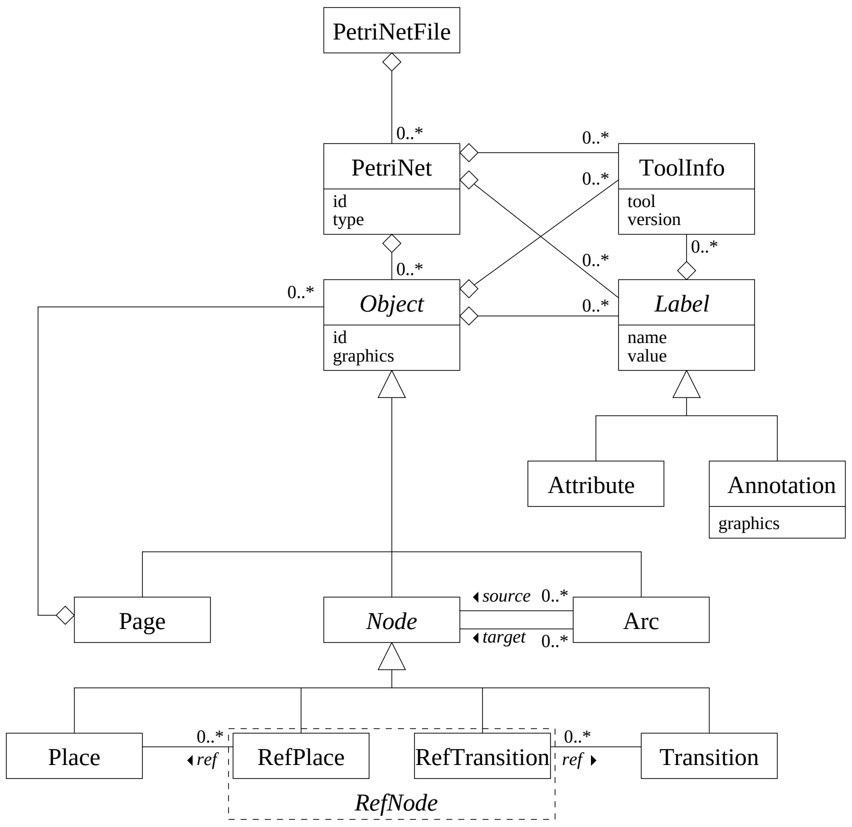

Fig. 1. Pure PNML: An overview

of a place, the guard of a transition, or the inscription of an arc. In addition, the Petri net itself may have some labels. For example, the declarations of functions and variables that are used in the arc inscriptions could be labels of a Petri net. The legal labels and the legal combinations of labels are defined by the *type* of the Petri net, which will be discussed in Sect. 2.3.

We distinguish between two kinds of labels – *annotations* and *attributes*. An annotation is a label with an infinite domain of legal values. Typically, a label will be displayed as text near the corresponding object. For example, names, initial markings, arc inscriptions, and transition guards are annotations. An attribute is a label with a finite (and small) domain of legal values. Typically, the value of an attribute is not display textually, but is represented in the form, style, or colour of the object itself. For example, the arc type could be an attribute of an arc with domain: `normal`, `read`, `inhibitor`, `reset` (and maybe some more). The shape of the arc will depend of this attribute. Another example is an attribute for classifying the nodes of a net as proposed by Mailund and Mortensen [8]. So, the basic difference between an annotation and an attribute of an object is that an annotation is displayed as a separate text near the object, whereas an attribute has impact on the shape of the object itself. Therefore, an annotation needs some information on the (relative) position of the displayed text, whereas attributes do not need this information. Note, however, that PNML does not define the impact of an attribute on the shape of the corresponding object. This is left to the implementation of the tool[3].

Graphical information. Each object and each annotation is equipped with some graphical information. For a node, this information is its position; for an arc, it is a list of positions that define intermediate points of the arc. For an annotation, the graphical information is its relative position with respect to the corresponding object[4]. Absolute as well as relative positions refer to the *reference point* of an object or of an annotation respectively. By default, the reference point is the middle of the graphical representation for an object; it is the lower left point of the graphical representation for an annotation. For an arc, the reference point is the middle of the first segment of the arc. Future extensions might allow us to define the position of a reference point of an object or an annotation explicitly. All positions refer to Cartesian coordinates (x, y). As for many graphical tools, the x-axis runs from left to right and the y-axis runs from top to bottom; but, we do not fix a cunit[5].

Tool specific information. For some tools, it might be necessary to store some internal information, which is not supposed to be used by other tools. In order to store internal information, each object and each label may be equipped with *tool specific information*. The internal format of the tool specific information is up to the tool. But, tool specific information is clearly marked and is assigned the name of the specific tool. Therefore, other tools can easily ignore this information. In general, we discourage the use of tool specific information. In some cases, however, tool specific information might be unavoidable.

Pages and reference nodes. A Petri net can be structured by the help of *pages* as known from several Petri net tools (e. g. Design/CPN [4]). A page is an object that may consist of other objects – it may consist even of further pages. An arc, however, may connect nodes on the same page only. In order to connect Petri net nodes on different pages, we can use *reference nodes*: A reference node may refer to any node of the Petri net – located on any page of the net. We require only that there are no cyclic references; this guarantees that, in the end, each reference node refers to exactly one place or exactly one transition of the Petri net. A reference node is only a representative for this node. Reference nodes

[3] Of course, the conventions for the use of some labels can recommend a graphical representation of objects with a particular attribute. But, this is not part of PNML.
[4] For an annotation of the net itself, the position is absolute.
[5] The size of objects and of labels as well as units are not part of PNML; but they can be easily included in a future version.

may have labels, too. But, these labels do not have any meaning. Concerning the semantics of the net, the reference node inherits the labels from the node it refers to. This way, it is always possible to flatten the corresponding net without knowing the meaning of the labels and without knowing the semantics of the particular Petri net type (see Sect. 4 for details).

2.2 Modules

Up to now, we have introduced the concepts of pure PNML, i. e. PNML without modules. In this section, we introduce the additional concepts of modular PNML, which are basically *module definitions* and *modules instances.* In order to illustrate these concepts, we start with an example.

Example. For simplicity, we use P/T-systems as the Petri net type in our example. Modular PNML, however, works for any Petri net type.

First, we consider the *definition of a module.* Figure 2 shows the definition of a module M1. This module consists of two places[6] x and y, two transitions t1 and

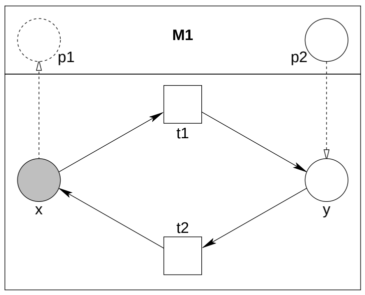

Fig. 2. A module M1

t2 and some arcs. This internal *implementation*, however, is not accessible from outside the module. In order to give the environment access to some internal elements of the module, the module defines an *interface.* In our example, the interface consists of a place p1, which is *imported* from the environment of the module, and a place p2, which is *exported* to the environment of the module. The import place p1 is a formal parameter, which is supplied when instantiating the module (see below for details); it is represented by a dashed circle. The export place can be used in the environment of an instance of the module; it is represented by a solid circle. The interface and the implementation of a module are related by references. In our example, reference place x, which is represented as a shaded circle, refers to import place p1. The reference is represented graphically by a dashed arrow and should not be confused with a Petri net arc, which would

[6] Actually, x is a reference place, which is indicated by the shading.

be represented by a solid arrow. So, the reference place x is a representative for a place that will be provided as a parameter when the module is instantiated. Likewise, the export place p2 refers to place y. So, a reference to export place p2 of an instance of the module will actually refer to place y (see below).

Next, we build a net from several instances of module M1. Figure 3 shows a graphical representation of a net n1 with three instances of module M1, which are named m1, m2, and m3, respectively. In the definition of the net, we define

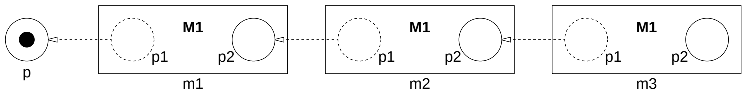

Fig. 3. A net n1 built from three instances of module M1

a place p with initial marking 1 and three instances of module M1. The first instance m1 takes the place p as the actual parameter for import place p1, which is represented graphically by a reference from the import place p1 of instance m1 to place p. The second instance m2 takes the export place p2 of m1 (denoted by m1.p2) as the actual parameter for p1. Likewise, the third instance m3 takes m2.p2 as the actual parameter for p1. Altogether, the net gives us the P/T-system shown in Fig. 4. We call this P/T-system the *semantics* of the net n1. This semantics will be defined, by recursively *inlining* the modules for the corresponding instances (see Sect. 4 for details). As expected, the three instances

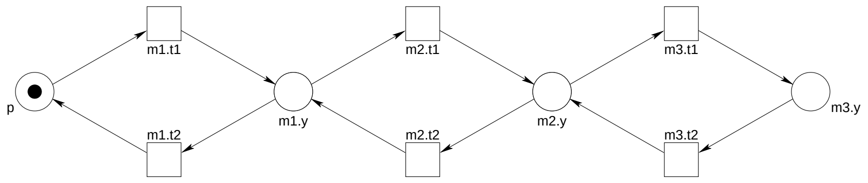

Fig. 4. The semantics of n1

form a line, which starts with place p and which merges p2 of an instance with p1 of the next instance. The names of the places and transitions in the different instances are qualified by the name of corresponding instance in order to avoid name clashes. Note that the import places and the export places have completely vanished; they are only representatives for the actual parameters.

Of course, the number of instances and their arrangement may be different in other nets built from module M1. For example, we could arrange the three instances of module M1 as a ring by passing m3.p2 as a parameter for p1 to the first instance. We will see in the definition of the semantics, that such cyclic use of export objects as parameters for import objects does not cause any problems.

This finishes our example, and we start with a more detailed discussion of the concepts of PNML.

Symbols. Sometimes, it is necessary to pass other arguments than nodes to a module. For example, a module could implement a channel for some type of messages, where the particular type is a sort provided by the environment when instantiating the module. This is known from templates in C++ or from parameterized data types, in general. Since modular PNML should be independent from a concrete Petri net type, we cannot fix a syntax for legal parameters for a module. But, we permit the definition of *symbols* – without knowing their meaning. Then, these symbols may be imported and exported in the same way as discussed for nodes in our example. Thus, symbols are objects, too. In particular, there are also *reference symbols*, which refer to other symbols. A symbol may occur within any label and must have a unique identifier.

In high-level Petri nets, the symbols could be sort symbols, operation symbols, and variable symbols, which define the legal inscriptions of places and arcs. By allowing to export and to import these symbols, it is possible to define such a symbol once and to use it in other modules.

Identifiers. In modular PNML, we restrict the values of identifiers to strings not containing a dot character (`.`). The reason is, that we need the dot for qualifying the name of an object of a particular instance of a module, as known from object oriented programming.

Module definitions. Simply spoken, a *module definition* is a net with an *interface*. For an instance of a module, only the objects of the interface are accessible from outside the instance. The rest of the module definition is its *implementation*.

Import and Export. The interface of a module contains objects, which can be accessed from outside the module. Remember that an object can be either a node or a symbol. We distinguish two kinds of interface objects: *import* objects and *export* objects. Import objects are representatives of objects that are provided as parameters upon instantiation of the module. The implementation may refer to these objects by a reference to the corresponding import object (cf. the reference from `x` to import node `p1` in Fig. 2). Export objects are defined inside the implementation of the module. Actually, an export object is just a reference object that refers to an object of the implementation of the module. This way, an export object allows the environment to refer to some object in the implementation of the module without knowing implementational details. For semantical reasons, however, modular PNML does not allow direct or indirect references from an export object to an import object of the module itself (see Sect. 4 for details).

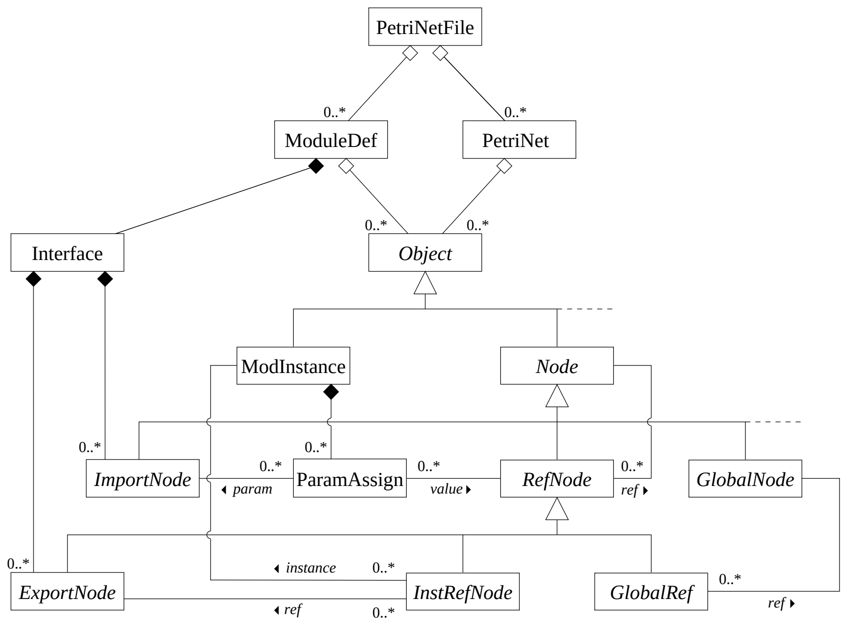

Fig. 5. Modular PNML: An overview

Global Nodes, Symbols, and References. Sometimes, we would like to have access to an object from all modules. We could define this object in the outermost net and pass it as a parameter to all other modules. This, however, is quite inconvenient. Therefore, modular PNML supports the definition of *global objects*, which can be referred to from any module without explicitly passing the global object as a parameter to that module. A reference to a global object is called a *global reference*. Note that nodes as well as symbols can be global.

Module Instances. A module can be used in a net (or in another module) by instantiating the module. This means that an instance defines a unique identifier within the net and assigns actual objects of the net to the parameters (i. e. to the import objects) of the module. Graphically, these assignments are represented by references from the import objects to its actual parameter. For example, import place `p1` of instance `m1` in Fig. 3 is assigned the actual parameter `p`. Export objects of a module instance are regarded as reference objects. This means that export objects can be used like reference objects. In particular, other reference objects can refer to them or they can be a parameter for an import object of a module instance.

Overview. Figure 5 gives an overview on the basic concepts of modular PNML. But, we leave out some details. For example, we do not distinguish places and

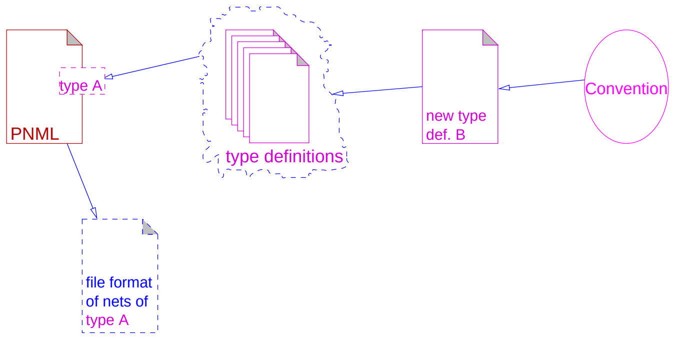

Fig. 6. Interplay of PNML, PNTD and the conventions document

transitions; we consider nodes only. Moreover, we omit symbols, since they exhibit the same structure as nodes.

2.3 Type definition

Up to now, we have discussed the general structure of a Petri net file. The available labels and the legal combinations of labels for a particular object are defined by a *Petri net type*. Technically, a Petri net type is a document that defines the XML syntax of labels; e. g. a Document Type Definition (DTD) file or a schema defined with an XML schema language such as XML Schema [13] or TREX [2]. Conceptually, a Petri net type is a specialization of the general format presented in the previous sections. It adds the definitions of the labels to the objects and to the net respectively.

Figure 6 illustrates the relation between PNML and a Petri net type definition (PNTD). A PNTD which is taken from a pool of various type definitions parameterizes the basic form of PNML. Thus, we get the PNML based file format for Petri nets of that type described by the PNTD. The right part of the illustration will be explained below.

2.4 Conventions

In principle, a Petri net type can be freely defined. In practice, however, a Petri net type chooses the labels from a collection of predefined labels that are provided in a separate document: the *conventions*. The conventions guarantee that the same label has the same meaning in all Petri net types. This allows us to exchange nets among tools with a different, but similar Petri net type.

The conventions are a collection of predefined labels. This collection, however, is not part of PNML. PNML provides only the mechanism for defining the conventions and for including parts of the conventions into a Petri net type.

Defining and maintaining the conventions document is an on-going process[7]. In Sect. 3.4, we discuss a small example, which illustrates how such a conventions document could look like. It may serve as a starting point to develop the conventions document. This development will converge in a document containing the most relevant labels from all kinds of Petri nets.

Figure 6 illustrates the relation between the conventions, the Petri net types, and the PNML. The right side shows the definition of a Petri net type based on the conventions, i. e. the labels are chosen from the conventions document. The new Petri net type definition is added to the pool of Petri net types and can be used as a net type in a PNML file.

3 Realization

In this section, we briefly present the PNML[8] syntax by discussing some examples. PNML is based on the Extensible Markup Language (XML) [12]. The Petri net, the objects, and the labels are represented as XML *elements*. An XML element is included in a pair of a start tag `<element>` and an end tag `</element>`. An XML element may have XML *attributes*[9] that equip the element with additional information. An XML attribute of an XML element is represented by an assignment of a value to a key (the attribute's name) in the start tag of the XML element `<element key=value>`. An XML element may contain text or further XML elements. An XML element without text or sub-elements is denoted by a single tag `<element/>`. In our examples, we sometimes omit some XML elements. We denote this by an ellipsis (`...`). All parts of PNML (PNML itself, the different PNTDs and the conventions) were implemented with the XML schema language TREX[10] [2].

The tags of the XML elements defined in PNML are named after the concepts (e. g. `<place>`, `<transition>` or `<page>`) given in Sect. 2. These tags of the concept are the keywords of PNML; they are called *PNML elements*. Labels, however are named after their meaning. Thus, any unknown XML element appearing in a Petri net or in an object can be clearly identified as a label of the net or the object. In our examples, the PNML keywords and the label for the name of an object are underlined. The tags of the other labels, however, are not underlined because they are keywords of a certain Petri net type definition not of the PNML itself.

[7] In fact, this process is strongly connected with the work of a standardization group that was founded at the 22nd Conference on Petri Nets (ICATPN) 2001 in Newcastle, U. K.

[8] Please refer to `http://www.informatik.hu-berlin.de/top/pnml/` for a full definition of PNML version 1.1 [6] and some examples.

[9] Do not confuse XML attributes with attributes of Petri net objects.

[10] TREX has been merged with RELAX to create RELAX NG [3]. TREX and RELAX NG are very similar such that the TREX implementation of PNML is its RELAX NG implementation, too. RELAX NG is specified by a committee of the XML standardization organization OASIS (`http://www.oasis-open.org`).

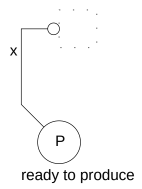

Fig. 7. An example net

3.1 The Petri Net Markup Language

Here, we discuss the syntax of the PNML concepts presented in Sect. 2.1. The first examples (List. 1-3) refer to the example net in Fig. 7.

The unique identifier of a Petri net or an object of a Petri net is given by an XML attribute `id` of the corresponding PNML element. The value of this attribute must meet the requirements for the attribute type `ID` of XML (cf. [12]); i. e. it must be a string starting with a letter or the underscore character, followed by letters, digits or several other characters except '.'.

Listing 1 shows the representation of a place with the identifier `p1`. The place has two labels; to be more precise, it has two annotations. The first one represents the name of the place `<name>`, whereas the second one represents its initial marking `<initialMarking>`. An annotation consists of its value `<value>` and, possibly, of some graphical information. In our example, both annotations as well as the place itself have graphical information, which are represented by XML

Listing 1. The PNML code of a place

```
<place id="p1">
  <graphics>
    <position x="20" y="40"/>
  </graphics>
  <name>
    <value>ready to produce</value>
    <graphics>
      <offset x="-10" y="10"/>
    </graphics>
  </name>
  <initialMarking>
    <value>P</value>
    <graphics>
      <offset x="-1" y="-1"/>
    </graphics>
  </initialMarking>
</place>
```

Listing 2. The PNML code of a transition

```
<transition id="t1">
  ...
  <toolspecific tool="PN4all" version="0.1">
    <hidden/>
  </toolspecific>
</transition>
```

Listing 3. The PNML code of an arc

```
<arc id="a1" source="p1" target="t1">
  <graphics>
    <position x="10" y="30"/>
    <position x="10" y="10"/>
  </graphics>
  <inscription>
    <value>x</value>
    <graphics>
      <offset x="-6" y="-16"/>
    </graphics>
  </inscription>
  <type value="inhibitor"/>
</arc>
```

elements `<graphics>`. The concrete definition of the XML element `<graphics>` depends on the context in which it appears. A place has a position, whereas an annotation has an offset position.

Listing 2 shows the representation of a transition, which is similar to the representation of a place. Transition `t1` contains tool specific information, which makes it a hidden transition of some imaginary tool PN4ALL version 0.1. Syntactically, toolspecific information is represented by `<toolspecific>`; this XML element must have at least the shown XML attributes and may contain further XML elements defined by the tool.

Listing 3 shows the representation of an arc: The source and the target node are given as XML attributes of the corresponding element `<arc>`. PNML requires that each arc has a unique identifier, which allows us to have two or more arcs between the same nodes. The graphical information of the arc contains a list of points. These points represent intermediate points of the arc. The offset in the graphical information of the `<inscription>` defines the position of the label relative to the reference point of the arc. Arc `a1` has an additional attribute called `<type>`; it indicates that it is an inhibitor arc. Note, that the shape of the inhibitor arc is tool dependent; other tools could use a different shape.

Listing 4. The PNML code of a page

```
<page id="pg1">
  <name>
    <value>Example page of the net</value>
  </name>
  <referencePlace id="rp1" ref="p1">
    <name>...</name>
    <graphics>
      <position x="20" y="20"/>
    </graphics>
  </referencePlace>
  <referenceTransition id="rt1" ref="t1">
    ...
  </referenceTransition>
  <place id="p2">...</place>
  <transition id="t2">...</transition>
  <arc id="a2" source="rp1" target="t2">
    ...
  </arc>
  ...
</page>
```

Listing 4 shows the representation of a page and of reference nodes of a Petri net. A page may have the same objects as the net itself – even pages and reference nodes. A reference node (indicated by tags `<referencePlace>` or `<referenceTransition>`) refers to a node of the net via the XML attribute `ref`. Its value refers to the identifier of a node of this net. Furthermore, a reference node may have its own graphical information, tool specific information, and labels. Remember that these labels have no real meaning, since they are ignored in the underlying flattened net. But, they allow reference nodes to carry their own name or other informal information. Note that the source and the target of an arc must be nodes on the same page.

Listing 5 shows the representation of a Petri net. A net consists of pages, modules instances (see below) and other objects. In our example, there is an annotation defining the net's name. The type of the net is given in the XML attribute `type`.

3.2 Modular PNML

Now, we discuss the syntax of modular PNML. To this end, we come back to the examples of Sect. 2.2. In order to keep the examples small, we omit graphical information from the PNML code.

Listing 5. The PNML code of a net

```
<net id="n1" type="HLnet">
  <name>
    <value>Example high-level net</value>
  </name>
  <place id="p1">
  ...
  </place>
  <page id="pg1">
  ...
  </page>
  ...
</net>
```

A module is defined by the PNML element `<module>`. This element contains both, the interface of the module and its implementation. The interface is tagged by `<interface>` and contains import and export objects of the module. The nodes of the interface may have graphical information as described above. The rest of the module contains the implementation of the module, which is the same as for nets. In addition, an implementation of a module may use instances of any other module. The only restriction is that there is no cyclic dependency between the modules. Moreover, reference objects of the implementation may refer to the import nodes of the module's interface.

Listing 6 shows the PNML code of module M1 in Fig. 2. There is a module with its interface and its implementation. The interface of a module contains nodes and symbols with their identifiers to be imported or exported. In our example (List. 6, cf. Fig. 2), there is one import place `p1` and there is one export place `p2`. The implementation part in our example does not use other modules. Note that reference objects of the implementation may refer to import objects but not to export objects of the interface of the module. Export objects refer to objects of the implementation. But, they are not allowed to transitively refer to import objects.

If a module M_1 (or a net) contains an instance of a module M_2, we say M_1 *uses* M_2. The use of a module is tagged by the PNML element `<instance>`. This element refers to the Uniform Resource Identifier (URI) of the corresponding module with the instance's XML attribute `ref`. As mentioned above, the uses relation must not have cycles. The PNML element `<instance>` contains references to nodes and symbols which serve as actual parameters for the import objects of the module. Such a reference names the parameter that is instantiated and refers to a 'real' object occurring in the instantiating net or module.

References to export objects of an instance are composed of a reference to that module instance (the XML attribute `instance`) and a reference to an export object of that instance (the XML attribute `ref`).

Listing 6. The PNML Code of the module in Fig. 2

```
<module name="M1">
  <interface>
    <importPlace id="p1"/>
    <exportPlace id="p2" ref="y"/>
  </interface>
  <referencePlace id="x" ref="p1"/>
  <transition id="t1"/>
  <transition id="t2"/>
  <place id="y"/>
  <arc source="x" target="t1"/>
  <arc source="t1" target="y"/>
  <arc source="y" target="t2"/>
  <arc source="t2" target="x"/>
</module>
```

Listing 7. The PNML Code of a net using modules (cf. Fig. 3)

```
<net id="n1">
  <place id="p">
    <initialMarking>
      <value>1</value>
    </initialMarking>
  </place>
  <instance id="m1" ref=URI#M1>
    <importPlace parameter="p1" ref="p"/>
  </instance>
  <instance id="m2" ref=URI#M1>
    <importPlace parameter="p1" instance="m1" ref="p2"/>
  </instance>
  <instance id="m3" ref=URI#M1>
    <importPlace parameter="p1" instance="m2" ref="p2"/>
  </instance>
</net>
```

Listing 7 shows the PNML code of the net in Fig. 3. Net `n1` contains a place `p` with an initial marking of one token and three instances of the module `M1`. Place `p` serves as the actual parameter for `p1` in instance `m1`. The module instance `m2` gets the export place `p2` of the instance `m1` as its actual parameter `p1` and so on.

Furthermore, PNML allows us to define global nodes and global symbols. They are tagged by `<globalPlace>`, `<globalTransition>`, and `<globalSymbol>` respectively. Similarly, we add the XML attribute `gref` to reference objects and

Listing 8. The PNTD for P/T-systems `ptNet.pntd`

```
<grammar ns="http://www.informatik.hu-berlin.de/top/pnml"
         xmlns="http://www.thaiopensource.com/trex">
  <include href="pnml.trex"/>
  <include href="conv.trex"/>
  <define name="NetType" combine="replace">
    <string>ptNet</string>
  </define>
  <define name="Place" combine="interleave">
    <optional>
      <ref name="InitialMarking"/>
    </optional>
  </define>
  <define name="Arc" combine="interleave">
    <optional>
      <ref name="Inscription"/>
    </optional>
  </define>
</grammar>
```

import parameters of interfaces for references to global objects. This XML attribute is alternative to both the XML attribute `instance` and `ref`. The value of the XML attribute `gref` refers to a globally defined object.

3.3 Petri net type definition

Next, we discuss the syntax of Petri net type definitions (PNTD). As mentioned above, we use the XML schema language TREX [2] for defining PNML and the particular PNTDs, as well. Listing 8 shows the PNTD for P/T-systems `ptNet.pntd`. For P/T-systems, we need two additional labels: one label for places, which represents the initial marking, and one label for arcs, which represents the arc inscription. Listing 8 shows the TREX file for the definition of P/T-systems. It starts with some TREX specific stuff for defining a *grammar*. Then it includes the definition of PNML and of the conventions document, from which we choose the definition of the labels. The conventions document itself, will be discussed in Sect. 3.4. Both, the PNML and the conventions document are TREX grammars, too.

Then, the file gives a name to the defined Petri net type (`ptNet`). Next, the file lists the objects that are equipped with additional labels. In our example, these objects are places and arcs (*"Place"* and *"Arc"*). The definition of a label can be either given explicitly, or can be taken from the conventions document. In our example, the labels are taken from the conventions document by referring to their names in that document (*"InitialMarking"* and *"Inscription"*). Technically,

Listing 9. An example conventions document

```
<grammar ns="http://www.informatik.hu-berlin.de/top/pnml"
         xmlns="http://www.thaiopensource.com/trex">
  <include href="pnml.trex"/>
  <define name="InitialMarking">
    <element name="initialMarking">
      <ref name="Annotation"/>
    </element>
  </define>
  <define name="Inscription">
    <element name="inscription">
      <ref name="Annotation"/>
    </element>
  </define>
</grammar>
```

this is achieved by the TREX element `<ref>`. The `<optional>` elements indicate that the labels are optional in this Petri Net type.

3.4 Conventions

At last, we give an example of a very simple conventions document, which defines just those two labels that were used in the example above. It should illustrate the mechanism of PNML for including definitions from a conventions document. A real conventions document would contain many more label definitions and would require a more involved taxonomy for labels. A standardized conventions document is subject to future research and to many discussions.

Listing 9 shows an example of entries in a conventions document. First, it includes the definition of labels (annotations in our case) from the general PNML definition `pnml.trex`. Then, it defines the two labels, which were used in the example above (*"initialMarking"* and *"inscription"*). Both of them are normal PNML annotations, which means that their value can be an arbitrary string; but we could use the full power of TREX for defining other domains.

4 Semantics

In this section, we will briefly discuss the semantics of PNML. Of course, it is impossible to define the semantics of the Petri net represented by a PNML file because this strongly depends on the Petri net type. Though there is some theoretical work on a unified framework for defining Petri nets, there is no formalism yet that deals with all types of nets definable by a PNTD. Therefore, we assume that along with a new PNTD, there will be a definition of the semantics of this Petri net type for nets without pages, references and modules.

In the following, we will give a precise semantics to pages, references, and modules by translating them to a net without these concepts. We proceed in two steps. First, we define a semantics for pure PNML by eliminating pages and references; then, we give a semantics for modular PNML by eliminating modules (resp. their instances) by translating them to pure PNML. For a more detailed discussion of this semantics, we refer to [6].

4.1 Pages and references

The semantics of pages and references is quite simple: We omit the pages from the PNML file and we resolve all references. Resolving references means that each reference object is replaced by the object it refers to (directly or indirectly via other reference objects). Actually, we delete the reference object, and we redirect all its attached arcs to the object it refers to. Note that by deleting the reference object, all labels of the reference object are lost, but this is no problem because labels of reference objects do not have any meaning.

Resolving all references is always possible, because PNML requires that there are no cyclic references between reference objects. This way, it is always possible to obtain an equivalent PNML file without pages and reference nodes. We have lost only graphical information and structuring information. We call this transformation *flattening* of a PNML net.

4.2 Modules

For modular PNML as defined in Sect. 2.2, we define the semantics by translating it to pure PNML. To this end, we basically replace each instance of a module by a copy of its implementation. In order to avoid name clashes, the identifiers of the implementation of a module will be preceded by the instance's name, where the names are separated by a dot[11]. In order to keep the module structure of the original model, the implementation of each module instance is defined on a separate page, which receives the name of the module instance. We call this replacement the *inlining* of a module instance.

The inlining process can be applied recursively for modules that use module instances in their implementation. In the end, we obtain a net in pure PNML. Note that inlining of module instances will never give us cyclic references, provided that there are no cyclic references in the module definition itself and no references from an export node to an import node. Thus, we can flatten the obtained net after the inlining process.

Figure 8 shows three stages of the inlining and flattening process for the net from Fig. 3. The first line shows the original net with three instances of module M1. The second line shows three different stages of the inlining process for the three different instances m1, m2, and m3: for m1, it shows the replacement by the module definition; for m2, it shows the replacement after renaming all objects

[11] By forbidding the dot in identifiers in modular PNML, this simple naming scheme gives us unique names for all instances, even when applied recursively.

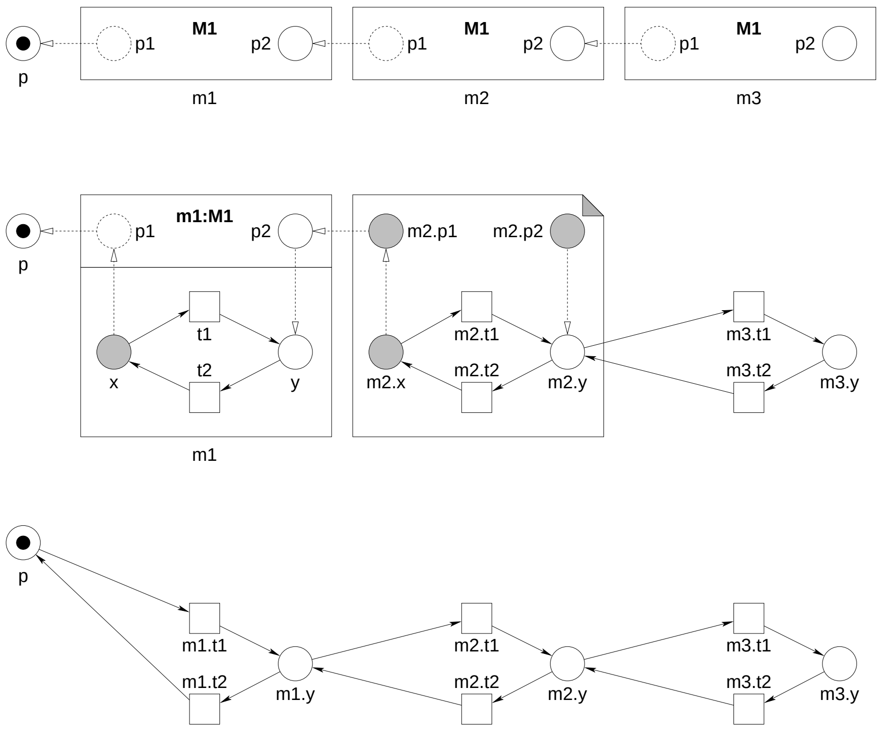

Fig. 8. Illustration of inlining and flattening

(by preceding each name with the instance's name) on a separate page; for m3, the reference nodes and the page have already been eliminated (flattening). The bottom line shows the overall semantics of the three module instances after inlining and flattening.

5 Conclusion

In this paper, we have presented PNML and its modular extension as an interchange format for Petri nets. The main feature of PNML is its universality, which is achieved by representing a net as a labelled graph along with a Petri Net Type Definition. The Petri Net Type Definition defines the legal labels for a particular Petri net type. The definition of modular PNML shows that universality carries over to a module concept, which can be used with any Petri net type. This way, PNML provides a way to exchange Petri nets between different tools without loosing too much information.

PNML itself is now in a stable state. Currently, it is supported by the PNK [10] and Renew [11]. For a broader acceptance, we need a standardized set of Petri net types and a standardized set of labels, which covers most of the existing versions of Petri nets. PNML provides a mechanism for defining Petri net types and for using labels from a conventions document. The Petri net types and the conventions document, however, are not part of PNML. The development and the standardization of the conventions document is an on-going process, which requires further research and interaction among researchers. More information on this process can be found on the web: `http://www.informatik.hu-berlin.de/top/pnml/`.

References

1. Rémi Bastide, Jonathan Billington, Ekkart Kindler, Fabrice Kordon, and Kjeld H. Mortensen, editors. *Meeting on XML/SGML based Interchange Formats for Petri Nets*, Århus, Denmark, June 2000. 21st ICATPN.
2. James Clark. TREX – tree regular expressions for XML. `http://www.thaiopensource.com/trex/`. 2001/01/20.
3. James Clark and Makoto Murata (eds.). RELAX NG specification. `http://www.oasis-open.org/committees/relax-ng/`. 2001/12/03.
4. Design/CPN. `http://www.daimi.au.dk/designCPN/`. 2001/09/21.
5. Ekkart Kindler and Michael Weber. The Petri Net Kernel – an infrastructure for building Petri net tools. *Software Tools for Technology Transfer (STTT)*, 3(4):486–497, 2001.
6. Ekkart Kindler and Michael Weber. A universal module concept for Petri nets. an implementation-oriented approach. Informatik-Berichte 150, Humboldt-Universität zu Berlin, June 2001.
7. Arndt Lüder and Hans-Michael Hanisch. A signal extension for Petri nets and its use in controller design. *Fundamenta Informaticae*, 41(4):415–431, 2000.
8. Thomas Mailund and Kjeld H. Mortensen. Separation of style and content with XML in an interchange format for high-level Petri nets. In Bastide et al. [1], pages 7–11.
9. The Moses Project. `http://www.tik.ee.ethz.ch/~moses`. 2002/03/04.
10. Petri Net Kernel. `http://www.informatik.hu-berlin.de/top/pnk/`. 2001/11/09.
11. Renew: The Reference Net Workshop. `http://www.renew.de`. 2002/03/04.
12. World Wide Web Consortium (W3C) (ed.). Extensible Markup Language (XML). `http://www.w3.org/XML/`. 2000/10/06.
13. World Wide Web Consortium (W3C) (ed.). XML Schema. `http://www.w3.org/XML/Schema`. 2001/05/02.