A Universal Module Concept for Petri Nets – an implementation-oriented approach –

Ekkart Kindler[1] and Michael Weber[2]

[1] Mathematisch-Geographische Fakultät, Katholische Universität Eichstätt, Germany
Ekkart.Kindler@ku-eichstaett.de
[2] Institut für Informatik, Humboldt-Universität zu Berlin, Germany
mweber@informatik.hu-berlin.de

Abstract The Petri Net Markup Language (PNML) is a proposal for an interchange format for Petri nets, which supports all kinds of Petri net types. New extensions to Petri nets can be easily defined as a new Petri net type.
In this paper, we propose *modular PNML*, which allows us to build Petri nets in a modular way. As PNML itself, modular PNML is independent from a particular Petri net type. The semantics of modular PNML is defined by a translation to pure PNML independently from the concrete Petri net type. Therefore, the module concept can be used with any tool that supports pure PNML or a future standard, which is currently evolving from it.

Keywords: Petri nets, interchange format, modules, XML, generic.

1 Introduction

Many concepts for defining Petri net modules and for building Petri nets from modules have been proposed in the literature. Most proposals, however, are restricted to a particular version of Petri nets. Some theoretical work has been done to unify these proposals and to extract the common principles (see [4] for an overview). These approaches, however, do not work for 'nasty' Petri net types—at least, it is hard to make them work. Moreover, these approaches are not so much interested in interchanging files among different tools and in a comfortable way for creating several instances of the same module.

In this paper, we propose a module concept that works for all Petri net types. Though simple and without a deep theory, this concept provides quite powerful means for building Petri nets in a modular way. Moreover, it has a clear semantics, which is independent from a particular Petri net type. The work on this concept was inspired by a talk of Shmuel Katz on the VeriTech project [7]. The VeriTech project aims at integrating many verification tools available in the Formal Methods community. In his talk, Shmuel Katz mentioned that they could not find an appropriate Petri net tool or an appropriate Petri net file format that could serve as an interface to the VeriTech project. Essentially, all formats were missing an appropriate concept of modularity. Therefore, Petri nets are not yet integrated into VeriTech.

So, we started to think on an appropriate format. In order not to exclude any Petri net type, we thought of a format that is independent from a particular Petri net type[1]. By the way, this independence from a particular Petri net type helped us to extract the essence of the module concept. In the end, we came up with a format which resembles the module concept of SMV [9], a well-known model checking tool (see NuSMV User Manual [2] for details). It supports the definition of modules that can be instantiated several times. Then, a system can be built from instances of modules. In contrast to SMV, our concept is tuned to Petri nets.

2 The Idea

Before going into the technical details, let us briefly illustrate the basic idea of the module concept. To this end, we use a simple example, where the Petri net type is classical P/T-systems.

First, we consider the *definition of a module* M1. The definition of this module is shown in Fig. 1. Module M1 consists of two places x and y, two transitions

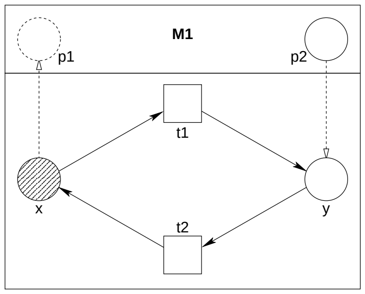


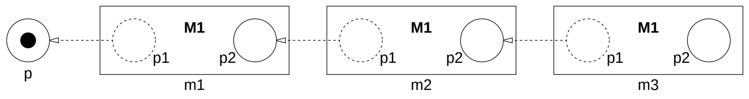
Fig. 1. A module M1

t1 and 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); 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

[1] Actually, we were working on a universal file format for Petri nets [6] at that time, but did not yet consider modularity.

by references. In our example, place[2] x refers to import place p1, which is represented by a dashed arrow from x to p1. So, the place x is a representative of the 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 place p2 of an instance of the module actually refers to place y (see below).

Next, we build a net from several instances of module M1. Figure 2 shows a graphical and a simple ad-hoc textual representation of a net n1 with three instances of module M1, which are named m1, m2, and m3, respectively. In the



```
def net n1:
{  place p: 1;
   m1 = instance M1(p1 = p);
   m2 = instance M1(p1 = m1.p2);
   m3 = instance M1(p1 = m2.p2);
}
```

Fig. 2. A net n1 built from three instances of module M1

definition of n1, we first define a place p with initial marking 1. Then, we define three instances of M1. The first instance m1 takes the previously defined place p as the actual parameter for p1. The second instance takes the export place p2 of m1 (denoted by m1.p2) as the actual parameter for p1. Likewise, the third instance takes m2.p2 as the actual parameter for p1. Altogether, the net n1 gives us the P/T-system shown in Fig. 3. We call this P/T-system the *semantics* of n1. This semantics will be defined, by recursively *inlining* the modules for the corresponding instances. As expected, the three instances form a line, which

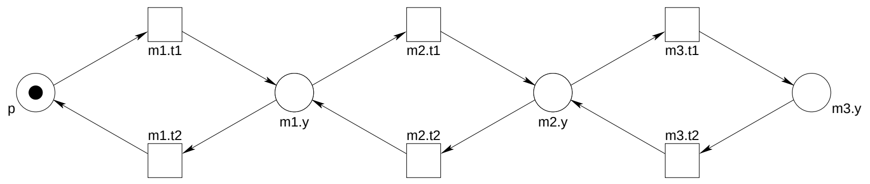
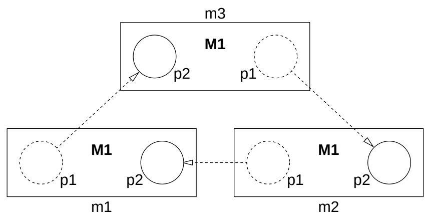


Fig. 3. The semantics of n1

[2] In the technical part, we will see that x is not a place. Rather, it is a reference place with a reference to the import place p1. This is the reason for the hatched representation of this place.

starts with place p and which links 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 corresponding instance in order to avoid name clashes. Note that the import 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 can arrange the three instances of module M1 in a circle (or a triangle) by passing m3.p1 as a parameter to the first instance as shown in Fig. 4. We will see in the formal definition of the



```
def net n2:
{  m1 = instance M1(p1 = m3.p2);
   m2 = instance M1(p1 = m1.p2);
   m3 = instance M1(p1 = m2.p2);
}
```

Fig. 4. Another net n2 built from three instances of module M1

semantics, that referring to the export place of an instance before the definition of this instance does not cause problems. Actually, the order in the textual representation does not play any role. The semantics of n2 is shown in Fig. 5.

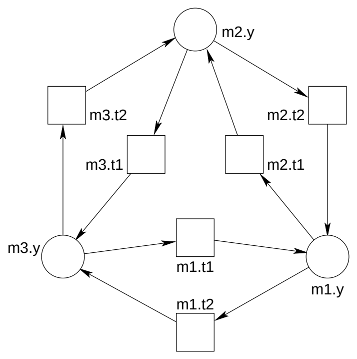


Fig. 5. The semantics of n2

This finishes the illustration of the module concept by the help of some examples. Of course, there are some more features, which will be discussed in the technical part. In particular, modules themselves may use instances of other modules in their definition provided that there are no cyclic dependencies. Moreover, a module may import and export also transitions and all kinds of symbols. The nature of the legal *symbols* and their meaning depend on the particular Petri net type. The module concept just provides a means to identify symbols such that they can be imported to and exported from modules.

In the above examples, we have used classical P/T-systems. The module concept, however, is not restricted to this Petri net type. It works for any Petri net type. Here, we are faced with the following problems:

1. We must have a format that captures any Petri net type.
2. We must define the semantics for the module concept independent from a particular Petri net type.

The first problem has been solved already by a generic XML based interchange format for Petri nets that, in principle, supports all Petri net types: the *Petri Net Markup Language* (*PNML*) [5]. Since the module concept is built upon this format, we rephrase the basic concepts and the XML syntax of the PNML in Sect. 3.

Then, in Sect. 4, we present the module concept and its XML syntax, which is called *modular PNML*. In order to give it a semantics, we define a mapping from modular PNML to pure PNML, which we call *inlining*. This inlining works independent from a particular Petri net type. This way, we do not need to know the semantics of the different Petri net types—we do not even need to know the syntactically correct Petri nets of this particular type. This solves the second of the above mentioned problems.

The inlining will be discussed in Sect. 4.3. Currently, we are developing a tool for this inlining operation for PNML resp. a standard interchange format currently evolving. This tool can be used in combination with any tool that supports the PNML or the future standard interchange format. Thus, the module concept can be used in combination with any of these tools.

3 Petri Net Markup Language

In this section, we give an outline of the current version of the *Petri Net Markup Language* (PNML) [5]. In this paper, we call it *pure PNML* in order to distinguish it from its modular extension, which will be presented in Sect. 4. In Sect. 3.1, we introduce the concepts of pure PNML; in Sect. 3.2, we give some examples.

3.1 The Concept

In the following, the concepts and the terminology of pure PNML are described. They are independent from any implementation of Petri nets or a concrete interchange format.

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 places, transitions, reference places, and reference transitions *nodes*, and we call a reference place and 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 marking of a place, the guard of a transition, or the inscription of an arc. The legal labels—and the legal combinations of labels—of an object are defined by the *type* of the Petri net, which will be defined later in this section. 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.

We distinguish between two kinds of labels: *annotations* and *attributes*. Typically, an annotation is a label with an infinite domain of legal values. For example, names, markings, arc-inscriptions, and transition guards are annotations. An attribute is a label with a finite (and small) domain of legal values. For example, an arc-type could be an attribute of an arc with domain: `normal`, `read`, `inhibitor`, `reset` (and maybe some more). Another example is attributes for classifying the nodes of a net as proposed by Mailund and Mortensen [8]. Besides this pragmatic difference, annotations have graphical information whereas attributes do not have graphical information[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 defines intermediate points of this 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 co-ordinates (x, y),

[3] Of course, the attribute arc-type with value `inhibitor` may affect the appearance of the corresponding arc. But, this effect is in the meaning of the attribute itself and does not come from some extra graphical information in the attribute.

[4] If it is an annotation of the net itself, the position is absolute.

where the x-axis runs from left to right and the y-axis runs from top to bottom; but, we do not fix a unit[5].

Pages and reference nodes. A Petri net can be structured by the help of *pages* which is known from several Petri net tools (e. g. Design/CPN [3]). 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 refers 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 may have labels. But, these labels do not have much 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 labels at all. Flattening means to merge each reference node to the node it refers to (directly or indirectly) and to ignore the labels of the reference nodes.

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—at least in the basic version of PNML.

Types and conventions. 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 or TREX.

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.

Up to now, the conventions define only those labels that are necessary for high-level nets. Defining further labels and maintaining the conventions doc-

[5] Size of objects and labels as well as units could be included in a future version.

ument is an on-going process. Here, we do not discuss these conventions; we provide the technique for defining new Petri net types and new labels only.

3.2 PNML by Examples

In this section, we briefly present the PNML[6] syntax by examples. PNML is based on XML [10]. The examples refer to the PNML version 0.99 [5]. In PNML, the net, the Petri net 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*[7] that equip an element with additional information. An XML attribute of an XML element is represented by an assignment of a value to a key (the attributes name) in the start tag of the XML element `<element key=avalue>`. XML elements 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 (`...`).

The tags of the XML elements defined in PNML are named after the concepts (e. g. `<place>`, `<transition>` or `<page>`) given in Sect. 3.1. 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.

The unique identifier of a Petri net or an object of a Petri net is always represented by an XML attribute `id` of the corresponding XML element. The value of this attribute must meet the requirements for the attribute type `ID` of XML (cf. [10]); i. e. it must be a string starting with a letter or the underscore character, followed by letters, digits or several other characters.

The first example (List. 1) shows the representation of a place with the identifier `p1`. The place has two labels, more precisely 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>`. Both, a place and an annotation may have graphical information represented by `<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.

A transition (List. 2) is represented in a similar way. Transition `t1` contains a tool specific information marking the transition as hidden in the example tool PN4ALL version 0.1. Syntactically, toolspecific information is represented by an

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

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

Listing 1. The PNML code of a place

```
<place id="p1">
  <graphics>
    <position x="10" y="20"/>
  </graphics>
  <name>
    <value>ready to produce</value>
    <graphics>
      <offset x="-5" y="2"/>
    </graphics>
  </name>
  <initialMarking>
    <value>P</value>
    <graphics>
      <offset x="0" y="0"/>
    </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="7" y="19"/>
    <position x="5" y="17"/>
  </graphics>
  <inscription>
    <value>x</value>
    <graphics>
      <offset x="-3" y="5"/>
    </graphics>
  </inscription>
  <type value="inhibitor"/>
</arc>
```

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>
```

XML element `<toolspecific>`; this element must have at least the shown XML attributes and may contain further XML elements defined by the tool.

An arc (List. 3) runs from a source node, referred to by the XML attribute `source`, to a target node (`target`). PNML requires that also each arc has a unique identifier. This simplifies the implementation of XML parsers. 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.

Listing 4 shows the representation of a page and of reference nodes of a Petri net. A page may have the same Petri net elements 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 etc. Note that an arc on a page starts and ends in nodes or in reference nodes defined on the same page.

The whole Petri net consists of pages and of Petri net elements. Listing 5 shows a net. It has an annotation defining the name of the net. The type of the net is available via the Uniform Resource Identifier (URI) given in the XML

Listing 5. The PNML code of a net

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

attribute `type`. In our case, `type` refers to a file that defines the legal labels and their syntax for the particular Petri net type. A description of this file, however, is beyond the scope of this paper.

4 Modular PNML

In this section, we introduce *modular PNML*, which equips PNML with a module concept. A module defines a building block from which other modules or Petri nets can be built[8]. A module consists of an interface and a module body. The interface defines those Petri net nodes that are visible from outside the module, whereas the module body defines the implementation of the module.

Section 4.1 introduces the module concept. Then, Sect. 4.2 describes the syntax of modules in PNML. And finally, Sect. 4.3 explains the semantics of the module concept.

4.1 The Concept

Now, we describe the concepts of the modular PNML. In the examples of this section, we refer to the figures (esp. Fig. 1, 2, and 4) in Sect. 2. The examples illustrate the module concept. There, we have seen that a module imports and exports some nodes. This way, it is possible for a module to refer to a node defined in its environment (the node passed as an argument for an import node when the module is instantiated). Likewise, the environment may refer to nodes of an instance of the module (the export nodes).

[8] Actually, a net is not built from modules, but from instances of modules. This allows us to use several instances of the same module to built a net.

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. These symbols may be imported and exported in the same way as nodes. Thus, symbols are objects, too. In particular, there are also *reference symbols* which refer to other symbols. A symbol may occur within any PNML element 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 import these symbols, it is possible to define such a symbol once and to use it in other modules.

Identifiers. For a reason that will become clear in Sect. 4.3, we restrict the values of identifiers (values assigned to the XML attribute `id`) in modular PNML to strings not containing a dot character (`.`).

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

Import and Export. The interface of a module contains nodes and symbols which can be accessed from outside the module. Remember that nodes and symbols are objects. We distinguish two kinds of interface objects: *import* objects and *export* objects. Import objects are representatives of nodes and symbols that are provided as parameters upon instantiation of the module. Export objects are defined inside the implementation of the module. Thus, an export object allows the environment to refer to some object in the implementation of the module without knowing implementational details. Import objects define an identifier such that objects actually defined outside the module and passed as a parameter upon instantiation of the module can be used inside the implementation of the module.

Global Nodes, Symbols, and References. Sometimes, it is conveniant to have global objects in a net, which can be used in all modules. To this end introduce the concept of *global objects* and *global references* (a reference to a global object). A global object can be referred to via a global reference from everywhere in the net even in modules and their instances without explicitly passing these objects as a parameter to these modules.

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. the import objects) of the module. Export objects of module instances are regarded as reference objects. This means that export objects can be used like reference objects. They become actual objects of the net.

Let us consider an example module as shown in Fig. 1. We added several graphical elements to the common Petri net notations in order to distinguish the special elements of a Petri net module. Both, the interface and the implementation of a module are placed into boxes. The identifier of a module is depicted in bold face inside the module interface. A node or a symbol of the implementation may refer to an import object of the interface. This representative of an object of the environment is depicted hatched. Likewise, an export object refers to a node or a symbol in the implementation such that an object in the environment referring to an export object of an instance of the module actually refers to that implemented node. In order to avoid cyclic references, an export object must not transitively refer to one of its import objects. References are depicted by dashed arcs.

In order to hide the implementation of a module we will draw the interface of a module without its implementation when instantiating a module. The interface of the module contains the interface objects and the module identifier.

When a module is instantiated some object must be assigned to each import object of the module. This is achieved by providing a reference to some object to the enclosing net or module for each import object of the instantiated module. Each instance is assigned a unique name which is depicted at the module interface. Then, the export objects of the instance can be referenced by a qualified names, i. e. by the name of export object preceded by the name of the instance (separated by a dot).

Figure 2 shows an example of an instance of a module in a net. This net consists of the place p (with one token in its initial marking) and the instances m1, m2, and m3 of the module M1 defined in Fig. 1. Figure 4 shows another example. Here, we dropped the place p because each export place of an instance serves as an import place for the next instance.

4.2 Syntax

Now, we explain how the module concept presented in Sect. 4.1 can be integrated into the Petri Net Markup Language (modular PNML) by extending pure PNML. We use the example nets of Sect. 2 to show the syntax of modular PNML. Our example PNML code does not contain graphical information in order to keep the examples small.

In modular PNML, we introduce the PNML element `<module>` which 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 in Sect. 3.

Listing 6. The PNML Code of the module in Fig. 1

```
<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>
```

The rest of the module contains the implementation of the module, these are the same elements as in a net of pure PNML. In addition, an implementation of a module may use instances of any other module. The only restriction is that there is no cyclic dependency.

Listing 6 shows the modular PNML code of module M1 in Fig. 1. 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. 1), 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 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 then we say M_1 *uses* M_2. The use of a module is tagged by the modular PNML element `<instance>`. This element refers to the Uniform Resource Identifier (URI) of the corresponding module with the instance's XML attribute `ref`. Remember, that the uses relation must not have cycles. The modular 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.

In modular PNML, 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`).

Listings 7 and 8 show the modular PNML code of the net in Fig. 2 and 4 respectively. Net `n1` (List. 7, cf. Fig. 2) contains a place `p` with an initial marking of one token and three instances of the module `M1`. Place `p` serves as the actual

Listing 7. The PNML Code of the net in Fig. 2

```
<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 8. The PNML Code of the net in Fig. 4

```
<net id="n1">
  <instance id="m1" ref=URI#M1>
    <importPlace parameter="p1" instance="m3" ref="p2"/>
  </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>
```

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. Net `n2` (List. 8, cf. Fig. 4) contains three instances of the module `M1`, such that the export place `p2` of one instance serves as the actual parameter for the import place `p1` of another instance.

Furthermore, modular PNML extends pure PNML with 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 import parameters of interfaces. 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.

4.3 Semantics

In this section, we define the semantics of modular PNML by translating it to pure PNML. This way, each net that is inductively built from instances of modules is translated to a Petri net without modules or instances of modules. In order to preserve the structure of the instances, each instance of a module is located on a separate page (with the name of the corresponding instance) and is related to the other instances by references.

The basic idea of this translation is bottom-up *inlining* all instances of modules. This bottom-up inlining is possible because the uses relation among modules is acyclic (and Noetherian). So we can start from those modules that do not use instances of other modules—we call these modules *basic modules*. In a module M that has only instances of basic modules, each instance is replaced by the corresponding basic module. This way, the module M becomes a basic module itself and can be inlined to the modules of the next level—and so on. Of course, we must take care that all objects of two different instances of a module carry different names; this can be achieved by an appropriate naming scheme and by consistently renaming all identifiers and references of a module, when inlining an instance of that module. This will be explained in more detail below.

Since we can apply the inlining operation recursively starting from the bottom, it is sufficient to discuss the inlining of a basic module to another module or net. Let us consider a module M in which an instance m of a basic module M1 occurs. Examples of two such modules are graphically represented in Fig. 6—for simplicity sake, we have only places in the interfaces of these examples.

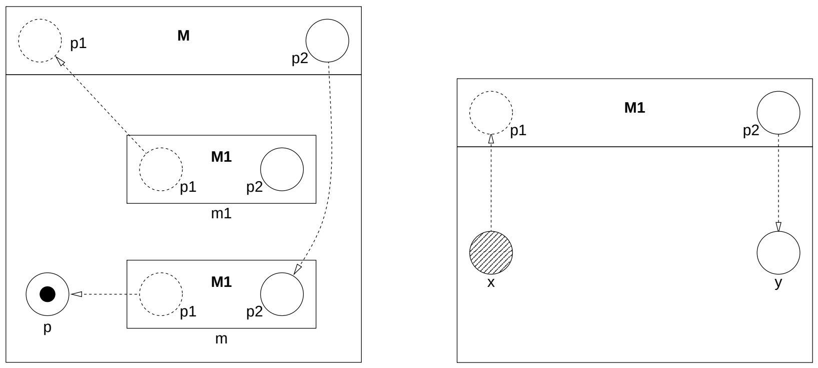


Fig. 6. A module M with an instance m of a basic module M1

For inlining instance m of module M1 into module M, we start with the module M1 as shown in Fig. 6 and transform it in the following steps:

Step 1 First, we convert each export object of M1 to a reference object. Each new reference object receives the same identifier and the same reference as the corresponding export object.

Step 2 Then, we consistently add a prefix 'm.' to each local identifier and to each local reference. The result of steps 1 and 2 is shown in Fig. 7. Note that

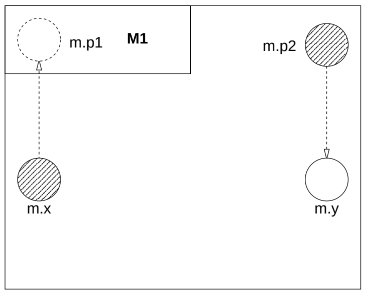


Fig. 7. Module M1 after applying steps 1 and 2

the identifiers of the import objects receive also the prefix m.

Step 3 Next, we replace the import objects by a reference object, were the identifiers remains the same. Each of these objects refers to the value that is passed as a parameter to this instance. The result is shown in Fig. 8. Note that these references are still outside of the module.

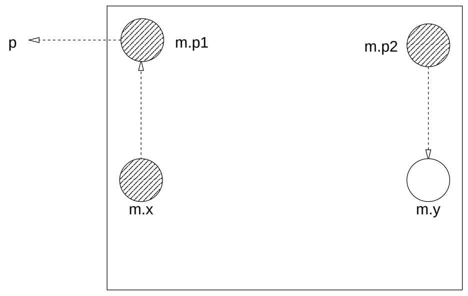


Fig. 8. Module M1 after steps 3

Step 4 At last, we place the resulting net in a separate page with identifier m and replace the instance m of module M1 in module M by this page as shown in Fig. 9

After these four steps, instance m is inlined into module M—there is no instance m of module M1 anymore. Rather, there is a page with the corresponding net. In the same way, we can inline all other instances into the module; in the end, module M is a basic module and can be inlined into modules with instances of M. Recursively, we can proceed all the way up in the module hierarchy until we reach the net in which these modules are used. Then, we can inline the basic

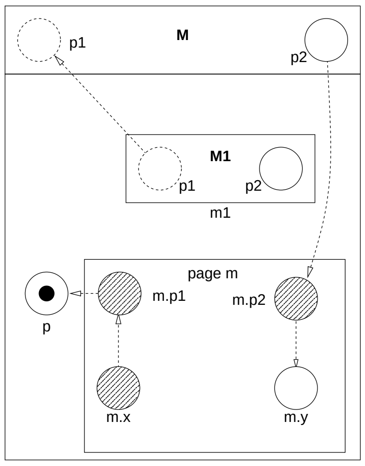


Fig. 9. Module M after steps 4

modules into this net, which gives us a net in pure PNML—without any instances of modules. Note that this net still consists of pages and has reference nodes. Eliminating pages and reference objects was already discussed in Sect. 3.1.

The consistent renaming of local identifiers and references guarantees that there are different instances of nets in the inlined version for different instances of modules in the original version. The only exception are global identifiers—these remain global by definition. Note that the acyclicity requirement for references is met in the constructed pure PNML net, because we do not allow references from export to import nodes and symbols.

5 Extensions

Sometimes, it is necessary to define many modules which are arranged regularly. For example, an n-bit adder can be composed from n one-bit adders, or a bounded communication buffer can be built from many one-message buffers in a row. When the number of such modules becomes larger, it is tedious to define each instance separately. In order to cope with this problem, modular PNML can easily be equipped with another feature, which allows us to create many instances at a time in an array.

Figure 10 shows an example. In the net n100 one hundred instances of module M1 are created at a time. The first instance m[0] receives the previously defined place p as an argument; the other instances m[i] receive the export place p2 of the immediately preceding instance m[i-1]. By the help of the case statement, we can deal with such irregularities. In the example, the first instance needs a special treatment. All others can be dealt with in the very same way. On the right hand side of the instantiation, we can refer to the other instances, where the arithmetic operations on i are modulo the number of created instances.

```
def net n100:
{ place p: 1;

  m[100] = case i:
             0       : m[i] = instance M1(p1 = p)
             [1..99] : m[i] = instance M1(p1 = m[i-1].p2)
           esac
}
```

Fig. 10. Example: Multiple instances

```
def net n1000:
{ place p: 1;

  m[1000] = case i:
             [0..999] : m[i] = instance M1(p1 = m[i-1].p2)
            esac
}
```

Fig. 11. Multiple instances

Another example would be a circle of thousand instances of module M1, which is shown in Fig. 11. Note that by interpreting the minus operation - modulo 1000 in this case, the export place of the last instance m[999] is linked to the import place of the first instance m[0].

The semantics and the inlining of multiple instances is, basically, the same as for a single instance. We must create the given number of instances. For each number i there is an instance with a unique identifier m[i]. We must take care, however, that arithmetic operations within the case statement are evaluated for each created instance, in order to provide the correct name for the referred instance. Remember that this evaluation is modulo the number of created instances.

For some technical reason, the characters for brackets, '[' and ']', cannot be used in XML identifiers. Therefore, m[i] reads m.i in the PNML file. This does not result in ambiguities in modular PNML, because digits are no legal first characters in identifiers of PNML and '.' is no legal character in identifiers of PNML.

Other possible extensions are still to be discussed. For example, it might be worthwhile to restrict the access to import or export places: An *input place* would exclude arcs from some transition of the module to this place; an *output place* would exclude arcs from the place to some transition of the module. This would allow us to rephrase different module concepts known from the literature. Note that input and output are orthogonal to import and export. An export place of a module could be an input place, or it could be an output place. Input and output gives us information on the direction of the token flow through a place. Export and import rather state whether the module or its environment provides the 'real' definition of this node.

6 Conclusion

In this paper, we have equipped PNML with a concept for building Petri nets in a modular way. In particular, this concept allows us to define modules that can be instantiated as many times as necessary in another module or a net.

Admittedly, the concept is simple and does not need a deep theory. Still, it comes with a universal semantics: i. e. a semantics that works with any Petri net type. Moreover, it provides all constructs necessary for constructing large systems from smaller components in a hierarchical way—including parameterized modules (known as templates in C++) and global definitions. Thus, the module concept resembles the way engineers build systems. In particular, the concept resembles the concepts used in SMV and other tools from formal verification. This way, a translation from and to these tools becomes feasible.

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. Alessandro Cimmatti and Marco Roveri. NuSMV 1.1. User manual, ITC-IRST and CMU, 1998.
3. Design/CPN. `http://www.daimi.au.dk/designCPN/`. last visited: 2000/11/13.
4. H. Ehrig, G. Juhas, J. Padberg, and G. Rozenberg, editors. *Unifying Petri Nets*, LNCS, Advances in Petri Nets. Springer, 2001, to appear.
5. Matthias Jüngel, Ekkart Kindler, and Michael Weber. The Petri Net Markup Language. In Stephan Philippi, editor, *7. Workshop Algorithmen und Werkzeuge für Petrinetze*, pages 47–52, Universität Koblenz-Landau, Germany, June 2000. AWPN. `http://www.informatik.hu-berlin.de/top/pnml/`.
6. Matthias Jüngel, Ekkart Kindler, and Michael Weber. Towards a generic interchange format for Petri nets. In Bastide et al. [1], pages 1–5.
7. Shmuel Katz and Orna Grumberg. VeriTech: Translating among specifications and verification tools. Technical report, The Technion, Haifa, Israel, March 1999.
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. Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
10. World Wide Web Consortium. Extensible Markup Language (XML), November 2000. `http://www.w3.org/XML/`.