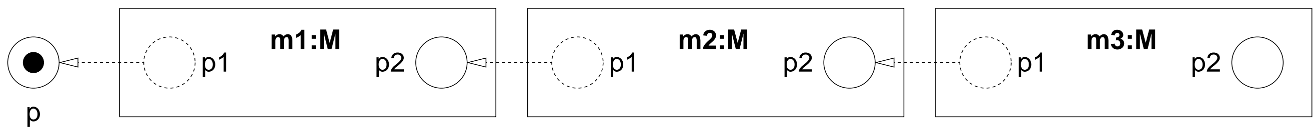
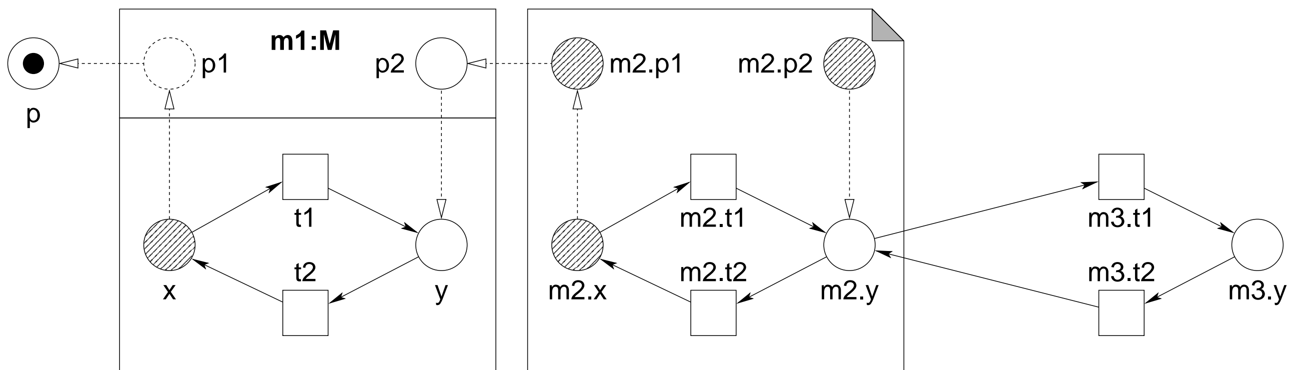
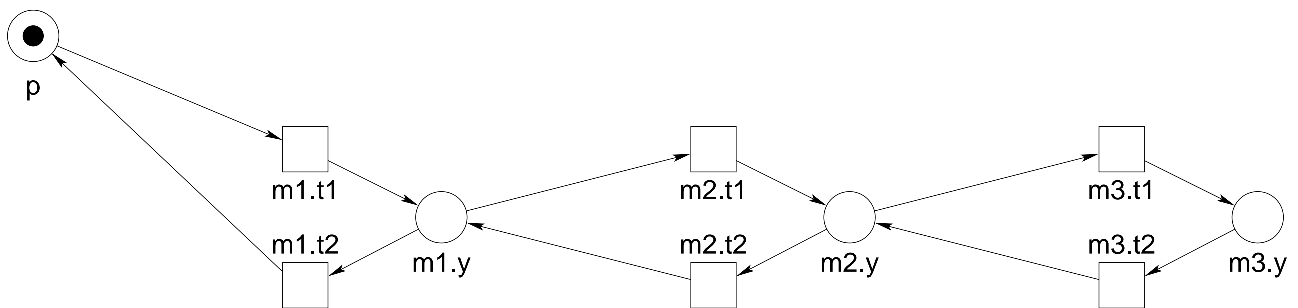


m1:M
p1
p2
m2:M
p1
p2
m3:M
p1
p2
p



m1:M
p1
p2
m2.p1
m2.p2
p
t1
t2
x
y
m2.t1
m2.t2
m2.x
m2.y
m3.t1
m3.t2
m3.y



p
m1.t1
m1.t2
m1.y
m2.t1
m2.t2
m2.y
m3.t1
m3.t2
m3.y

Cover Picture Story

Modules in Pictures

Ekkart Kindler, Technische Universität München, Institut für Informatik
kindler@informatik.hu-berlin.de

Michael Weber*, Humboldt-Universität zu Berlin, Institut für Informatik
mweber@informatik.hu-berlin.de

The cover picture is a graphical illustration of the module concept underlying *modular PNML*: an extension of the *Petri Net Markup Language* (*PNML*) with a module concept that works for all kinds of Petri net types [2].

The top of the cover picture shows a sequence of three *instances* m1, m2, and m3 of module M, where the *module definition* can be found in Fig. 1. The nodes at the *interface* of the instances of the module are related by so-called *references* in order to arrange the instances in a row. The middle of the cover picture shows different stages of the expansion of these instances into an overall model, which defines the semantics of the module concept. From left to right, the different stages are:

1. The instance m1 of module M is equipped with its *implementation* as shown in Fig. 1.

2. In the next stage, the implementation of instance m2 is converted into a separate page. On this page, all names are consistently renamed by preceding them with the instance's name. Moreover, the nodes of the interface of the module are converted to so-called *reference places*, which can be used as a representative for a place on a different page. This way, places on different pages can be connected without drawing Petri net arcs between them. In order to distinguish references from Petri net arcs, *references* are drawn as dashed arrows.

3. In a last step, the page border of instance m3 is omitted, references are resolved, and all reference places of this page are deleted. Note that resolving the references connects transitions m3.t1 and m3.t2 directly to m2.y.

The bottom of the cover picture shows the obtained overall model, where all steps have been applied to all instances. A more detailed discussion of the module concept and its

*Supported by the Deutsche Forschungsgemeinschaft within the project "Petri Net Technology"

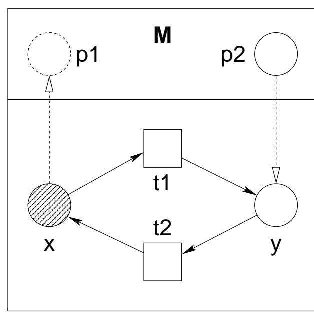


Figure 1: Definition of module M

semantics can be found in [2]. Here, we restrict ourselves to a brief introduction of its main ingredients. In particular, we discuss the simple trick which makes the module concept universal.

Pages and reference nodes

Pages are a well-known technique for editing and displaying large Petri nets. Maybe, the use of the page concept became popular by the tool DesignCPN. Of course, the parts of the nets on the different pages are not completely isolated. Therefore, there must be some way to relate the elements on different pages. This can be achieved by 'merging' places or by 'merging' transitions of different pages. We use *reference nodes* for this purpose. Graphically a reference node is represented by a shaded place or transition, respectively. Each reference node refers to another node on the same page or on another page[1]. Note that a reference node may refer to another reference node. Cyclic references, however, are excluded.

Modules

Pages allow us to draw large Petri nets. But, pages do not provide an abstraction mechanism by themselves[2]. Moreover, it is difficult to reuse the same net pattern in different contexts.

Modules will provide both, an abstraction mechanism and a reuse mechanism. A *module* can be instantiated several times and it can be used without knowing its internal details: For using a module, we need its interface only. We will discuss this mechanism in some more detail below. To this end, we must distinguish between the *module definition* and the different *module instances*.

The module definition consist of the *interface* and the *implementation*. The interface consists of *import nodes* and *export nodes*[3]. This interface is graphically shown at the top of the module definition. Import nodes are indicated by a dashed line whereas export nodes are indicated by a solid line. For the implementation of the module, an import node represents

[1] In our examples, we have indicated these references by dashed arrows. In a tool, however, these arrows would be difficult to draw because the nodes may be on different pages. Therefore, a reference will be by means of unique identifiers in a tool.

[2] Of course, pages could be used in a very disciplined way that provides some abstraction. But this discipline is left to the user.

[3] In our example, we have import and export places only. The concept, however, allows us to use places and transitions.

a node from outside the module, which must be provided for each instance of the module (indicated by a dashed arrow from the import node to the corresponding node outside the instance as shown on the cover picture). The implementation of the module may refer to an import node by references (see Fig. 1). Vice versa, an export node refers to a node of the implementation of the module. This way, this node can be accessed from outside the instance by a reference to this export node.

Note that an export node may refer to reference nodes of the implementation or to export nodes of some of its instances, too. But in order not to run in semantical problems, an export node must not refer to an import node (see [2] for details).

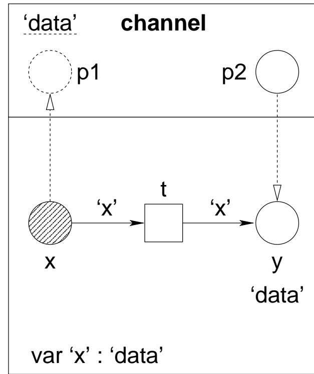


Figure 2: Definition of module channel

Symbols and reference symbols

Up to now, the module concept considers the net structure only. In many cases, however, we would like to import and to export other information to and from a module. For example, think of a module channel, which transmits data of some sort data. Now, we would like to use this module with different data types, integer and string. In order to achieve this, the information on the data type must be passed to the module upon instantiation of the module.

The kind of information, its syntax, and its semantics, however, depends on the particular Petri net type. In order to make the module concept universal, i. e. to make it work for any Petri net type, we must think of the common principle of all Petri net types: Any Petri net of any type, can be considered to consist of places, transitions, and arcs with some labels[4]. The labels themselves can be considered to be composed from symbols. The available symbols and the way in which symbols may be composed to legal labels depends on the Petri net type. Fortunately, we don't need this type dependent information for making the module concept work. We need to identify the used *symbols* only.

Then, we can define *reference symbols*, *import symbols*, and *export symbols* analogously to reference nodes, import nodes, and export nodes. Even the semantics can be defined in the same way.

For example, Fig. 2 shows the definition of the module channel, where the occurrence of a symbol within a label is indicated by quotation marks. Symbol data is an import symbol,

[4]Actually, this is the only assumption underlying the PNML.

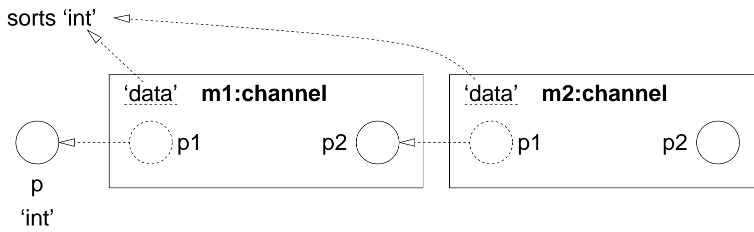


Figure 3: Two instances of module channel

which is indicated by a dashed underline. Now, we can instantiate this module: Fig. 3 shows a sequence of two instances of channel, where the symbol data refers to sort int. Figure 4 shows the semantics, which is obtained in the same way as on the cover picture by

1. adding the implementation of the module for each instance, by

2. converting each implementation into a page, where all nodes and symbols are consistently renamed by preceding them with the instance's name, and by

3. omitting the page borders and by resolving the references.

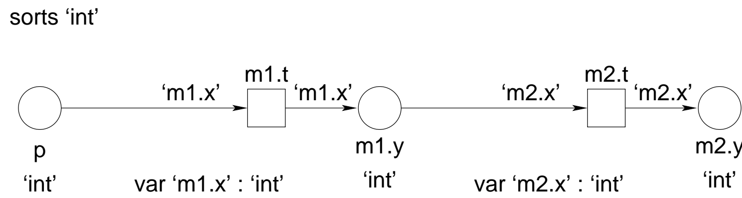


Figure 4: Semantics of the model from Fig. 3

Conclusion

We have sketched a simple module concept which works for all Petri net types provided that symbols occurring in the labels of the Petri net are clearly marked. Combined with the PNML [1], an XML-based interchange format for Petri nets, it gives us modular PNML [2].

References

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

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