Operational PNML: Towards a PNML Support for Model Construction and Modification

João Paulo Barros[1,2]*and Luís Gomes[1]
{jpb,lugo}@uninova.pt
[1]Universidade Nova de Lisboa/UNINOVA
Campus da FCT, P-2829-516 Monte de Caparica – Portugal
[2]Instituto Politécnico de Beja – Portugal

Abstract

The *Petri Net Markup Language* (PNML) allows the specification of Petri net models based on their primitive elements: places, transition, and arcs. This paper proposes a complementary way to define Petri net models, based on a set of operations on nets. This approach allows the construction and modification of Petri net models in a highly flexible way supporting not only modular composition, but also model modifications.

1 Introduction

It is a well-known fact that Petri net models are often difficult to use in practice due to the problem of rapid model growth. This fact has been the subject of numerous proposals that are able to reduce the graphical models' size (e.g. [1,4–8,10,11,15]). Those proposals are in fact abstraction constructs allowing information hiding: the subnets hide part of their inner details. Here, we propose the use of two generic operations for the specification of model composition and modification, named *net addition* and *net subtraction*. The operations are proposed as a complementary way for the definition of Petri net Models by the Petri net Markup Language (PNML) [3,12,16,18]: besides

*Work partially supported by a PRODEP III grant (Concurso 2/5.3/ PRODEP/2001, ref. 188.011/01).

the exhaustive enumeration of all Petri net elements, we get a way to define new nets by operations on existent nets. Net addition (one of the operations) can support the two well-known approaches to model's construction, namely:

1. top-down model construction by net refinements
2. bottom-up model construction by net aggregation (specifying environments for the existent model).

These two composition types are commonly found in literature, even if with different names and presentations, and are implemented in numerous tools.

From an engineering perspective, we can think of a third type of net modification supported by the net addition operation: as in point 2 above, the model is modified by immersing it in another model; yet, this is not seen as a bottom-up construction where one model module grows by connecting it to another, but as the modification of one or more existent model modules by a single new module. This new module imposes structural and behavioural modifications to all the modified modules. This corresponds to the realisation, at the net level, of crosscutting requirements [1, 13, 17]. To this end we can use the proposed operations, which can compose, in an orthogonal way, the existing nets or modules.

Several tools specify the structure definition of the whole model by annotations added to the net components. These annotations establish the connection among the several pages or modules. For example: node fusion is specified by annotating one node with the identifier of another node in the same or in a different subnet.

Differently, we propose a total separation between the composition information and the net components annotations. A new net can be defined by describing the way other nets are related. This net definition is based on operations that refer to the set of operand nets. This allows the quick specification of a large net model without any kind of graphical editing: we simply specify the textual expressions composing the nets; the original nets definitions remain exactly the same.

The proposed operations are both amenable to a simple textual representation to be made available by tools. This allows the representation, in a compact and readable format, of a large number of net compositions. These can be seen as contributing either to system development or to future modification of a "completed" system.

The net addition and net subtraction operations, here presented, generalise a previous version [1] by supporting non disjoint fusion sets, and operate

on net instances as presented in [9] and formally defined elsewhere [2]. They are currently restricted to low-level nets. Future work will investigate their generalisation to high-level nets.

The following section briefly presents where our proposal fits in the PNML concept. Thereafter, the operations are defined while presenting illustrative examples.

1.1 From PNML to Operational PNML

The *Petri Net Markup Language* (PNML) defines Petri nets based on an exhaustive enumeration of all the primitive Petri net elements. For example, for the net in Fig. 1a [1] we have the PNML in Listing 1. More specifically, this PNML code is called *basic PNML* to distinguish it from *Structured PNML* [3, 12, 16, 18]. The latter uses pages and reference nodes.

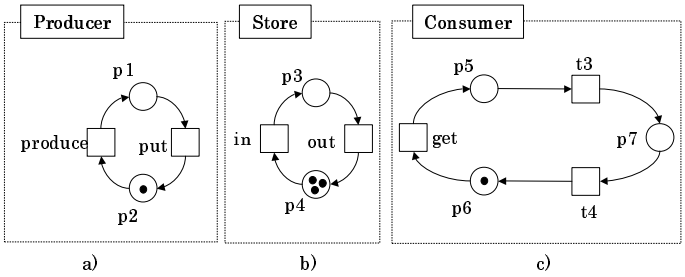

Figure 1: a)Net Producer; b)Net Store; c)Net Consumer.

Listing 1: PNML code for Producer net in Fig. 1a

```
<pnml xmlns="http://www.informatik.hu-berlin.de/top/pnml/ptNetb">
  <net id="Producer" type="http://www.informatik.hu-berlin.de/top/pntd/ptNetb">
    <place id="p1">
      <initialMarking>
        <text>0</text>
      </initialMarking>
    </place>
    <place id="p2">
      <initialMarking>
        <text>1</text>
      </initialMarking>
    </place>
    <transition id="produce">
      <name>
        <text>produce</text>
      </name>
    </transition>
    <transition id="put">
```

[1] The dotted rectangles enclosing each net in Fig. 1 have no special meaning: they are only used to better identify a single net as defined in the PNML specification.

```
            <name>
               <text>put</text>
            </name>
         </transition>
         <arc id="a1" source="produce" target="p1">
            <inscription>
               <text>1</text>
            </inscription>
         </arc>
         <arc id="a2" source="p1" target="put">
            <inscription>
               <text>1</text>
            </inscription>
         </arc>
         <arc id="a3" source="put" target="p2">
            <inscription>
               <text>1</text>
            </inscription>
         </arc>
         <arc id="a4" source="p2" target="produce">
            <inscription>
               <text>1</text>
            </inscription>
         </arc>
      </net>
   </pnml>
```

We will not use graphical information in the PNML code (e.g. the PNML *graphics* element) as it is irrelevant for the purpose of this paper. We also do not rely on any specific Petri net Type definition (PNTD): the (to be presented) operations only deal with the net structure. Yet, we do use the Place/Transition PNTD for testing purposes.

Currently, there are two PNML based ways to compose Petri net models:

1. Using *Structured PNML*: each net becomes a page and each page can then reference the other pages through the use of place references, transition references, or both.

2. Using *Modular PNML* [14,18]: each net becomes a module. Each module can then specify an implementation part and an interface part, with import nodes (reference nodes), output ports (a subset of the *true*, non reference, nodes). The modules are connected through the import and export nodes, only.

Structured PNML offers a simple way to partition a single net model in several sub-models referencing each other: each node in a single page can be a reference to another node in another page. We thus have nodes and *reference nodes*. The resulting model fuses each reference node with a real node resulting in a single real node. The modeller still has to know the unique large model, resulting from all the references among pages.

Although translatable to structured PNML, modular PNML is quite different as it allows the creation of several independent instances of a given net

model (the module). In this sense modular PNML fulfils the usual reusability criteria that each module should be usable in several distinct contexts, possibly at the same time, by means of multiple instances. Furthermore, the module designer does not have to know the contexts where the module is going to be used, but only the module functionality.

In this paper we propose an additional approach where each net can be defined in any of the two following forms:

1. The net is defined by exhaustive enumeration of all its elements, using basic PNML.

2. The net is defined as a set of operations on other nets and, possibly, their instances.

The first form corresponds to the use of basic PNML: the net has no reference nodes or any other construct able to specify some form of connection or interface to other nets or pages. The use of *Structured PNML* or *Operational PNML* also seems a straightforward, and potentially useful, "natural" extension for the first form.

Complementarily to the first form, in the second form, the net is specified through a set of operations on existent nets and net instances. The defined net is the result of one, and only one, operation in the set: the one that *return*s it.

Using a set theory analogy, we can say the first form defines nets *by extension* (listing all its elements), while the second form defines a Petri net *by comprehension*.

We call the basic PNML complemented with the second form for net definitions *Operational PNML*.

Using Operational PNML it is possible to define Petri net models as an arbitrarily complex sequence of operations on any number of other net instances. These operations include net addition, a generalisation of net composition by node fusion, but also subtraction allowing the inverse modifications. Other operations can certainly be defined but we restrict to these, as we believe they are highly generic, intuitive, and offer an implicit structure for all modifications (either additive, or subtractive) as their operands are also nets.

The following section illustrates, in an informal and example-based way, the use of the Operational PNML concept through the presentation of the two referred operations.

2 The Operational PNML specification

This section details the already presented second form for net specification. This is achieved by means of an example based approach for each of the proposed net operations. The examples show the corresponding Operational PNML code. The section ends by briefly presenting the RELAX NG grammar for Operational PNML.

We start by informally defining net vectors, net instances, and node indexes. After, we present fusion sets followed by the proposed operations. Finally, we highlight the main differences to the basic PNML grammar.

2.1 Net instances and node indexes

When using Operational PNML, each net defined as in basic PNML (e.g. Listing 1) is also seen as a template from which other identical nets, named *net instances*, can be generated. These instances are always seen as elements in a *net vector*. For example, from net *Store* (see Fig. 1b) we can generate a net vector with three elements 1 to 3. These vector elements (or net *Store* instances) are denoted by suffixing the respective *instance number* to the net *Store identifier*. For example, for net vector $Store[1 \ldots 3]$ we get three nets: $Store[1]$, $Store[2]$, and $Store[3]$. Additionally, all nodes in a net instance need to refer the net instance as a prefix. For example, nodes in net *Store[1]* initially identified by the respective identifier (*id*), become identified by the net instance *1* followed by the identifier, with a dot as a separator: $Store[1].in$, $Store[1].p3$, $Store[1].p4$, and $Store[1].out$.

As exemplified latter, it is also potentially useful to refer to a set of net nodes by index. For this reason, we also propose the use of *indexes* in the *nodeID*s. The index is made part of the node identifier (*nodeID*) and appears between square brackets: for example, nodes *out* of *Store* instances can be referred by *(Store[i].out)* where i is the index and, in this case, also an iterator variable between to given values: *first* and *last* (see Listing 6 for an example). This does not imply a modification to the *node.content* define in the PNML grammar, but it does imply processing the *nodeID* attribute when iterators are used.

2.2 Fusion sets

A set of nodes to be fused is called a *fusion set*. These are of two kinds: *placeFusion* (when all elements of the set are places) and *transitionFusion* (when all elements of the set are transitions). In Operational PNML, each node to be fused is specified by referring the net instance (it belongs to) and

the node identifier. Each fusion set will correspond to a single node in the resulting net. Hence, there is a *nodeID* attribute allowing the specification of an identifier, possibly containing indexes for the generated node. Listing 2 shows the Operational PNML RELAX NG grammar for the *placeFusion* and *transitionFusion* specifications.

Listing 2: The transitionFusion and placeFusion elements in the Operational PNML RELAX NG grammar.

```
<define name="transitionFusion.element">
   <element name="transitionFusion">
      <ref name="transitionFusion.content"/>
   </element>
</define>
<define name="placeFusion.element">
   <element name="placeFusion">
      <ref name="placeFusion.content"/>
   </element>
</define>
<define name="transitionFusion.content">
   <attribute name="nodeID">
       <data type="token"/>
   </attribute>
   <oneOrMore>
       <ref name="transitionRef"/>
   </oneOrMore>
</define>
<define name="placeFusion.content">
   <attribute name="nodeID">
       <data type="token"/>
   </attribute>
   <oneOrMore>
       <ref name="placeRef"/>
   </oneOrMore>
</define>
```

We can also specify fusion set vectors: *placeFusionVector* and *transitionFusionVector* (see Listing 3). As the name implies, a fusion set vector is a compact notation for a list of fusion sets. Each fusion set vector has an associated iterator variable (*iterator* attribute) and an associated range specified by the *first* and *last* attributes. It also has the *nodeID* attribute to name the generated nodes. The list of fusion sets is generated by the iteration of the iterator variable across the values in the range. Each iteration corresponds to a fusion set with the respective value for the iterator variable. The index for the generated nodes can be the iterator variable (or even a function of it). This allows the generation of multiple nodes with distinct indexes (see Listing 6 for an example).

Listing 3: The fusion set vector element in the Operational PNML RELAX NG grammar.

```
<define name="placeFusionVector.element">
   <element name="placeFusionVector">
      <attribute name="iterator">
```

```
            <data type="NMTOKEN"/>
          </attribute>
          <attribute name="first">
            <data type="nonNegativeInteger"/>
          </attribute>
          <attribute name="last">
            <data type="nonNegativeInteger"/>
          </attribute>
          <ref name="placeFusion.content"/>
        </element>
      </define>
      <define name="transitionFusionVector.element">
        <element name="transitionFusionVector">
          <attribute name="iterator">
            <data type="NMTOKEN"/>
          </attribute>
          <attribute name="first">
            <data type="nonNegativeInteger"/>
          </attribute>
          <attribute name="last">
            <data type="nonNegativeInteger"/>
          </attribute>
          <ref name="transitionFusion.content"/>
        </element>
      </define>
```

Next we present, through examples, the net addition and the net subtraction operations. Both use fusion sets and fusion set vectors.

2.3 Operations

We now present the two proposed operations. The use of operations for the definition of Petri net models by Operational PNML is based on the following six points:

1. At least one net must be defined as in basic PNML (*by extension*). Although certainly possible, we do not propose any way to define a net in the absence of primitive nets.

2. The operations, defining a new net, operate on a specified set of net instances (the operand nets).

3. We can define vectors of nets instances based on existent nets.

4. Net instances are nets identified by the identifier (*id*) of the originator net and a non negative integer called the net instance number.

5. The nodes in net instances are identified by the net *id*, the net instance number, and the *id* of the original node.

6. The result of each operation is a net, which can be used in another operation. The set of operations defining a net is partially ordered, as

the result of some operations can be an operand for other operations. There must be one, and only one, operation that returns the defined net. This is specified in the proposed PNML extension by giving the name "return" to the resulting net.

Next, we present the two operations by referring to examples of the respective PNML code.

2.3.1 Net Addition

Net addition without fusion sets defines a disjoint union of two nets: they remain disconnected. Listing 4 shows the disjoint union of nets Producer and Store.

Listing 4: Net defined as a disjoint union of two other nets (net addition operation).

```
<pnml xmlns="http://www.informatik.hu-berlin.de/top/pnml/ptNetb">
   <net id="ProducerAndStore"
        type="http://www.informatik.hu-berlin.de/top/pntd/ptNetb">
      <operations>
         <addition result="return">
            <net netID="Producer"/>
            <net netID="Store"/>
         </addition>
      </operations>
   </net>
</pnml>
```

This first Operational PNML example already illustrates the main distinctive points compared to basic PNML:

1. Only the inside of the net element is (radically) changed; it now contains, exclusively, a new element named *operations*. This element can contain any number of operations. In the presented example, it contains a single operation identified by the element *addition*, which includes a list of operand nets (nets *Producer* and *Store* in the example). This nets are defined as separated basic PNML files with the same name as the one specified by the *netID* attribute.

2. All operations have an attribute named *result*. This attribute specifies the name of the resulting net. Inside the *operations* element one and only one operation must give the name *return* to the respective *result* attribute; that operation defines the net. In Listing 4 we have a single operation and thus it must return the respective result by specifying "return" as the value for its *result* attribute. But, for example, in the net in Listing 6, the first addition result is net *PS*, which is latter used by the other addition.

The net defined in Listing 4 is illustrated in Fig. 2a. As we defined an empty fusion set (no node fusions), net addition becomes a disjoint union. Note that, for presentation purposes, Fig. 2a shows the *nodeID*s, while an editor would show the respective text in each node's *name* label.

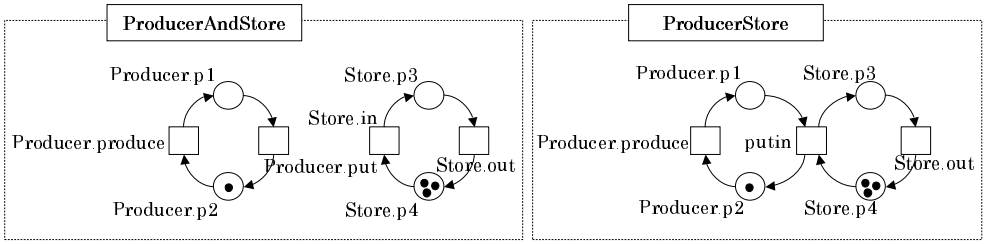

Figure 2: a)Net defined in Listing 4; b) Net defined in Listing 5.

Usually, net addition specifies fusion sets. After net union, some nodes are fused according to the specified fusion sets. Listing 5 shows the specification of the *ProducerStore* net. This is, again, the result from the addition of nets *Producer* and *Store* but now the transitions *Producer.put* and *Store.in* are fused. The result is shown in Fig. 2b [2]. Notice the difference to net addition without node fusion in Fig. 2a.

Listing 5: Net defined as an addition operation

```
<pnml xmlns="http://www.informatik.hu-berlin.de/top/pnml/ptNetb">
   <net id="ProducerStore"
        type="http://www.informatik.hu-berlin.de/top/pntd/ptNetb">
      <operations>
5        <addition result="return">
            <net netID="Producer"/>
            <net netID="Store"/>
            <merge>
               <transitionFusion nodeID="putin">
10                <transition netID="Producer" nodeID="put"/>
                  <transition netID="Store" nodeID="in"/>
               </transitionFusion>
            </merge>
         </addition>
15    </operations>
   </net>
</pnml>
```

Listing 6 defines a new net through the specification of two net addition operations. The first addition returns the net *PS*, a composition of net *Producer* with three instances of net *Store*: for each value of i between 1

[2]The resulting net is shown only for illustration purposes as typically we do not want, or need, a graphical representation.

and 3, the *Producer.put* transition is fused with transition *Store[i].in*, returning a transition *putin[i]*. This is shown in Listing 6. Again for illustration purposes, the resulting net (*PS*) is graphically shown in Fig. 3a.

Note the use of a *fusion set vector* (specified by the *fusionSetVector* element).

The net *PS* is then composed with three instances of net *Consumer*, now using an iterator over the *Store[i].out* nodes indexed by the respective net instance index (the *i* in *Store[i]*). This returns the net *PSC* shown in Fig. 3b.

Listing 6: A net defined by two net additions.

```
<pnml xmlns="http://www.informatik.hu-berlin.de/top/pnml/ptNetb">
  <net id="PSC" type="http://www.informatik.hu-berlin.de/top/pntd/ptNetb">
    <operations>
      <addition result="PS">
        <net netID="Producer"/>
        <netVector netID="Store" first="1" last="3"/>
        <merge>
          <transitionFusionVector nodeID="putin[i]"
                                  iterator="i" first="1" last="3">
            <transition netID="Producer" nodeID="put"/>
            <transition netID="Store[i]" nodeID="in"/>
          </transitionFusionVector>
        </merge>
      </addition>
      <addition result="return">
        <net netID="PS"/>
        <netVector netID="Consumer" first="1" last="3"/>
        <merge>
          <transitionFusionVector nodeID="outget[i]"
                                  iterator="i" first="1" last="3">
            <transition netID="PS" nodeID="Store[i].out"/>
            <transition netID="Consumer[i]" nodeID="get"/>
          </transitionFusionVector>
        </merge>
      </addition>
    </operations>
  </net>
</pnml>
```

Eventual modifications on the net *label*s in specific PNTDs must be known by the tool handling the Operational PNML files. This includes, for example, the resulting net marking after place fusion. More specifically, the tool should provide a simple way for the PNTD designer to define the transformations on each net label. For example, it should be possible to easily specify a marking addition for interface places in net addition and a marking subtraction for interface places in net subtraction. This reverse operation, is presented next

2.3.2 Net Subtraction

Net subtraction allows the removal of undesirable net parts. Inside a methodology based on iterative development, this modification also allows the re-

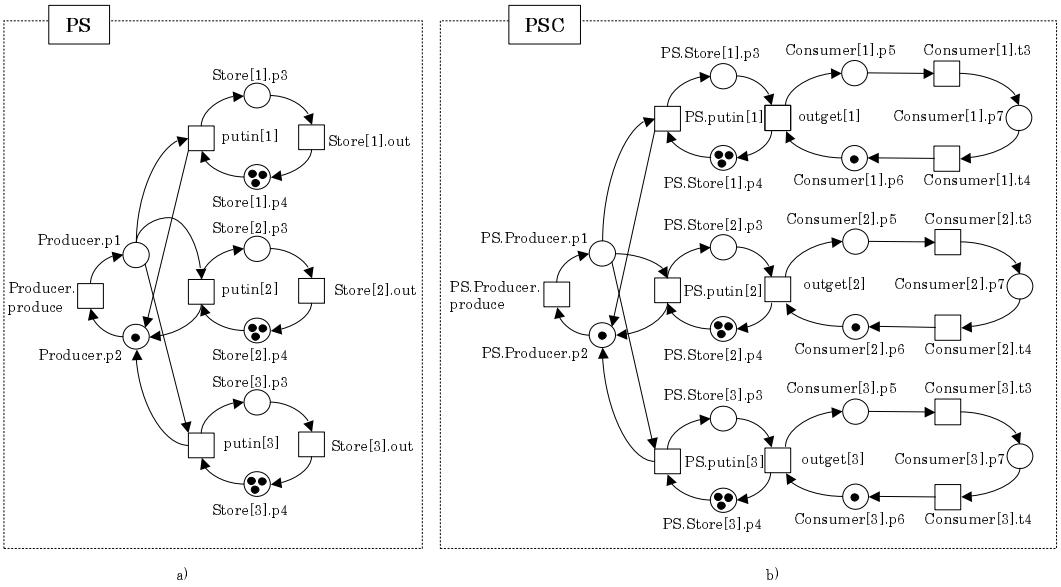

Figure 3: a)Net *PS* in Listing 6; b) Net in Listing 6.

placement of a specific part by newer or updated models. This reverse modification is potentially very useful: for example, given a net A to which we add a net B, net subtraction allows the return to the initial net A, by subtracting the net B. This is also the expected use for net subtraction.

Listing 7 defines a net by subtracting net *Store* from net *ProducerStore*. This effectively undoes the previously presented addition (Listing 5): the result (net *Producer2* in Listing 7) is the same net as the *Producer* net (Fig. 1a), although with some distinct *nodeID*s. Listing 7 also illustrates the use of a *transitionFusion* containing a single transition. This effectively allows the renaming of the referenced transition's *nodeID* by the *transitionFusion*'s *nodeID*.

Listing 7: Net defined as a subtraction operation

```
<pnml xmlns="http://www.informatik.hu-berlin.de/top/pnml/ptNetb">
   <net id="Producer2"
   type="http://www.informatik.hu-berlin.de/top/pntd/ptNetb">
      <operations>
         <subtraction result="return">
            <net netID="ProducerStore"/>
            <net netID="Store"/>
            <merge>
               <transitionFusion nodeID="put">
                  <transition netID="ProducerStore" nodeID="putin"/>
                  <transition netID="Store" nodeID="in"/>
               </transitionFusion>
               <transitionFusion nodeID="produce">
                  <transition netID="ProducerStore" nodeID="Producer.produce"/>
               </transitionFusion>
```

```
            </merge>
            <removal>
               <placeFusion nodeID="p3">
                  <place netID="ProducerStore" nodeID="Store.p3"/>
                  <place netID="Store" nodeID="p3"/>
               </placeFusion>
               <placeFusion nodeID="p4">
                  <place netID="ProducerStore" nodeID="Store.p4"/>
                  <place netID="Store" nodeID="p4"/>
               </placeFusion>
               <transitionFusion nodeID="out">
                  <transition netID="ProducerStore" nodeID="Store.out"/>
                  <transition netID="Store" nodeID="out"/>
               </transitionFusion>
            </removal>
         </subtraction>
      </operations>
   </net>
</pnml>
```

Net subtraction is simply net addition followed by the removal of a set of nodes. The set of nodes to be removed are also specified by fusion sets, but inside the *removal* element. Each arc connected to a node to be removed, is also removed.

2.4 Operational PNML Grammar

The support for the presented net operations by the PNML grammar implies one minimal modifications to the the basic PNML RELAX NG grammar: the addition of an alternative to the net element content (*define name="net.content"* in the RELAX NG grammar). Together with the net labels, we can have either the usual net definitions, or the operations element. This extra possibility is specified by the addition of a single $<$ *choice* $>$ pattern. The new grammar is defined by merging it with the basic PNML grammar file through the include mechanism in RELAX NG. Only *net.element* is overridden with minimal modifications. All the remainder basic PNML grammar remains the same.

The complete Operational PNML grammar in RELAX NG, specifying the syntax for all the presented operations, is available at `http://www.uninova.pt/gres/opnml/`. A sample script, allowing the translation from Operational PNML to PNML, is also available, together with the presented examples.

3 Conclusion

The presented operations offer an alternative and generalised way to specify Petri net models. Their support by PNML does not imply any change to the present PNML definition, as all the operations are readily definable

by an extended grammar, which still recognises all valid basic PNML files. Also, the definition of an *Operational Structured PNML* and an *Operational Modular PNML* seems straightforward. This would allow the use of operational definitions in an orthogonal way to the existent modular compositions. This is particularly interesting for introducing modifications spanning several existent modules.

It is perfectly possible to define other simpler and more primitive operations, like renaming of nets and nodes, and creation and removal of nodes and arcs. All are potentially useful, but due to their unstructured nature, great care must be taken to avoid unmanageable models.

Future work will generalise the presented Operational PNML semantics to handle dependencies among the labels in the fused nodes and the respective arcs' inscriptions. This is especially significant for some high-level nets' classes where the modification of a node label can imply modifications to the respective arcs' inscriptions.

Acknowledgements The authors thank the two anonymous reviewers whose comments helped to improve the paper significantly.

References

[1] João Paulo Barros and Luís Gomes. Modifying Petri net models by means of cross-cutting operations. In *Proceedings of the 3rd International Conference on Application of Concurrency to System Design*. IEEE Computer Society, jun 2003.

[2] João Paulo Barros and Luís Gomes. Net model composition and modification by net operations: a pragmatic approach. In *Proceedings of the 2nd IEEE International Conference on Industrial Informatics (INDIN'04)*, Jun 2004. to appear.

[3] Jonathan Billington, Søren Christensen, Kees van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri net markup language: Concepts, technology, and tools. In W. van der Aalst and E. Best, editors, *Proceeding of the 24th International Conference on Application and Theory of Petri Nets*, volume 2679 of *LNCS*, pages 483–505, Eindhoven, Holland, jun 2003. Springer-Verlag.

[4] Wilfried Brauer, Robert Gold, and Walter Vogler. A survey of behaviour and equivalence preserving refinements of Petri nets. *Lecture Notes in Computer Science; Advances in Petri Nets 1990*, 483:1–46, 1991.

[5] Peter Buchholz. Hierarchical high level Petri nets for complex system analysis. In Valette, R., editor, *Lecture Notes in Computer Science; Application and Theory of Petri Nets 1994, Proceedings 15th International Conference, Zaragoza, Spain*, volume 815, pages 119–138. Springer-Verlag, 1994.

[6] Søren Christensen and N. D. Hansen. Coloured Petri nets extended with channels for synchronous communication. *Daimi PB-390*, 1992.

[7] Søren Christensen and Laure Petrucci. Modular analysis of Petri nets. *Computer Journal*, 43(3):224–242, 2000.

[8] Rainer Fehling. A concept of hierarchical Petri nets with building blocks. In *Proceedings of the 12th International Conference on Application and Theory of Petri Nets, 1991, Gjern, Denmark*, pages 370–389, June 1991.

[9] Luís Gomes and João Paulo Barros. On structuring mechanisms for Petri nets based system design. In *Proceedings of the 2003 IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2003)*, pages 431–438. IEEE Catalog Number: 03TH8696, sep 2003.

[10] Xudong He and John A. N. Lee. A methodology for constructing predicate transition net specifications. *Software–Practice and Experience*, 21(8):845–875, August 1991.

[11] P. Huber, K. Jensen, and R. M. Shapiro. Hierarchies in coloured Petri nets. In *Proceedings of the 10th International Conference on Application and Theory of Petri Nets, 1989, Bonn, Germany*, pages 192–209, 1989.

[12] M. Jüngel, E. Kindler, and M. Weber. The Petri net markup language. In S. Phillipi, editor, *Workshop Algorithmen und Werkzeuge fr Petrinetze*, oct 2000.

[13] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, 11th *European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer Verlag.

[14] Ekkart Kindler and Michael Weber. A universal module concept for Petri nets. In *Proceedings des 8.Workshops Algorithmen und Werkzeuge fr Petrinetze / Gabriel Juhas und Robert Lorenz (Hrsg.) – Katholischen Universität Eichstätt, 2001*, pages 7–12, 1-2 October 2001.

[15] Julia Padberg. Petri net modules. *Transactions of the SDPS*, 6(3):121–196, sep 2002.

[16] Petri net markup language (PNML). `http://www.informatik.hu-berlin.de/top/pnml/about.html`, 2004.

[17] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. IEEE Computer Society Press, 1999.

[18] Michael Weber and Ekkart Kindler. The Petri net markup language. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technology for Communication Based Systems*, volume 2472 of *LNCS*, pages 124–144. Springer-Verlag, 2003.