On the Use of Programming Languages for Textual Specification of Petri Net Models

João Paulo Barros[1,3] and Luís Gomes[2,3]

[1] Instituto Politécnico de Beja, Escola Superior de Tecnologia e Gestão, Portugal
[2] Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, Portugal
[3] UNINOVA, Portugal
{jpb, lugo}@uninova.pt

Abstract. [4] As a general interchange format for Petri net models, the Petri Net Markup Language (PNML) allows the specification of Petri net models for all Petri net classes. Those models are typically generated by graphical editors for each Petri net type. Yet, there is no general way to specify Petri net models in a human-friendly textual notation. Instead of proposing a standard for such textual notation, this paper proposes the use of popular general purpose programming languages for the creation and modification of net models defined using PNML. To that end, the paper presents a model for the concepts, and the respective interrelations, that should be available to define Petri net models in a compact textual format. After, it presents a general framework to specify model composition, using node fusion, for any Petri net class. The framework allows the specification of node fusions and node refinements based on the specification of fusions for each node and net label. The labels' fusions are defined through the implementation of an abstract data type for the respective Petri net type definition. This allows a general support for model structuring, where several well-known graphical conveniences, e.g. node references and synchronous channels, can be supported and seen as particular cases.

1 Introduction

Petri nets are well-known as a set of graphical specification languages. In fact, the most useful and identifying Petri nets characteristics are clearly represented in a graphical way, namely active and passive entities, parallelism, synchronization, and resources allocation and consumption. Yet, it is also a well-known fact that graphical models can easily become too large to be readable or even to be built using a graphical editor. One common solution to this problem is the use of hierarchical structuring mechanisms. Yet, these large models, and also many other smaller models, can be faster, and more conveniently, built using a text editor rather than a large, and sometimes complex, graphical editor. Hence, we present a proposal for the creation of human-friendly textual specifications using general purpose programming languages. These specifications are used together

[4] This paper revises and extends a previous paper from the same authors [1].

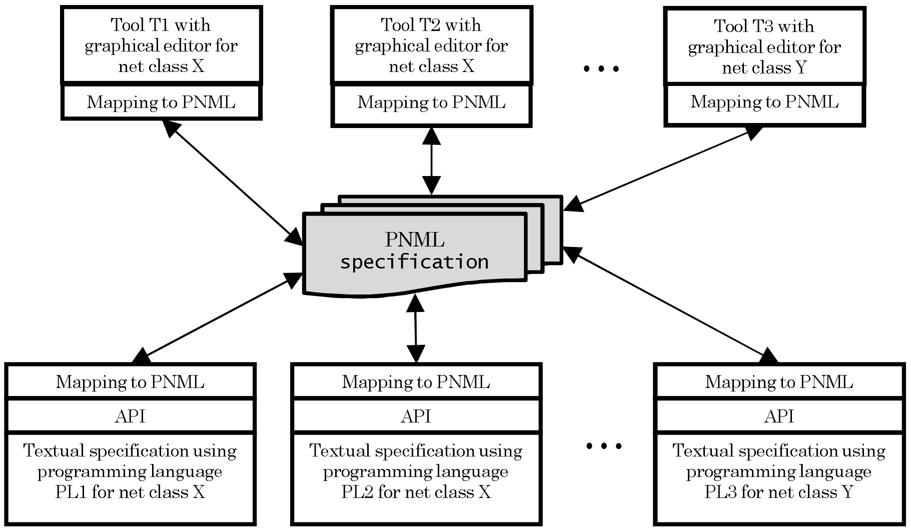
with Petri Net Markup Language (PNML) [2] specifications. Hence, they support any kind of Petri nets specifiable using PNML.

We do not present a unique concrete syntax for a human-friendly textual language for two reasons: (1) we believe that would have to be the result of a large consensus in the context of a standardization process; (2) it seems unlikely that such language would be preferred over the use of existing popular programming languages. Instead, we present the components for an abstract syntax that should cover the specification of models in any Petri net class. The semantics is also informally presented. Presently, it is clearly desirable that this effort takes PNML abstract syntax as a foundation, as the PNML already provides a generalized way to specify Petri net models defined in any Petri net class. PNML is based on XML, which is a good choice for a specification language that was designed to be read and written by tools. Yet, XML does not have a human friendly syntax. In particular, it is not as readable or as versatile as popular programming languages. XML is meant to structure data, not to process it. Hence, besides PNML, we believe Petri net modelers can significantly benefit from a readable textual language with processing capabilities and additional and alternative structuring constructs. In particular, the synergies between processing capabilities and additional structuring constructs, found in most programming languages, allow more compact specifications and several layers of abstractions. For this reason, our proposal for the creation of human-readable textual languages is, in fact, a proposal for the creation of high-level Application Programmer Interfaces (APIs) for general purpose programming languages, preferably object-oriented languages. Those APIs must support the reading and writing of PNML models. This means we can read, write, and modify PNML specifications using a programming language where the API is available. The end result should always be a standard compliant model. In this sense, each programming languages together with the respective API acts as an alternative to the tools, typically graphical editors, that also read and write standard compliant specifications (namely PNML files). This is illustrated in Fig. 1.

We should stress that textual modeling is a complementary alternative to graphical models. In software engineering, programs and many specifications are usually thought as text and for good reasons, namely the compactness and generality it allows. For example, SDL [3] one of the most widely used formal languages in industry has a graphical and also a textual notation. In fact, graphical languages have well-known advantages but also limitations (e.g. [4] and [5]). The use of an API for a full blown programming languages makes the textual specifications extremely versatile. Basically, the APIs give the modeler options: for each net one can choose a graphical editor to construct a graphical model or, if visual information is not as important as rapid prototyping or integration, a textual specification can be built.

This paper has two contributions:

1. To propose the use of high-level programming languages to create and modify nets based upon the elements present in PNML, namely, net, place, transition, and arc. More specifically, we propose a model, specified as a UML



Tool T1 with graphical editor for net class X
Mapping to PNML
Tool T2 with graphical editor for net class X
Mapping to PNML
Tool T3 with graphical editor for net class Y
Mapping to PNML
PNML specification
Mapping to PNML
API
Textual specification using programming language PL1 for net class X
Mapping to PNML
API
Textual specification using programming language PL2 for net class X
Mapping to PNML
API
Textual specification using programming language PL3 for net class Y

Fig. 1. PNML as the interchange format for graphical and textual model specifications

class diagram [6], for the implementation of APIs showing the entities and their inter-relationships. This model should be seen as an abstract syntax for the high-level API allowing the programming language to be used for the following tasks:

(a) To read existing PNML models, modify them and write them back to disk. If convenient, the absent graphical layout information can be automatically generated.

(b) To define net models and write them to disk. These models can be created by enumerating all its elements, or as the result of the net operations (the second contribution).

(c) To define the net classes, more specifically, the label names for each of those elements and the semantics of node fusion based on labels fusion (the "fusion engine").

2. A minimal set of operations that allow the modeler to compose Petri net models. Hierarchically and horizontally. These operations are intuitive as they are based on node fusion, the most obvious and common way to compose net models.

Regarding the first contribution, we identify a set of concepts and constructs that should be supported by any API. This use of a programming language is proposed as a complement not only to specifications expressed in XML (PNML), but especially to graphical specifications. Besides, programming languages provide a large set of structuring constructs and data processing capabilities, which can be put to good use when creating repetitive models, doing rapid prototyping, or integrating models with other tools.

Using text, models can be constructed either by enumeration or, more interestingly, as the result of more compact specifications made possible by the use of a minimal set of operations (part of the API) and constructs usually available in programming languages. This can mean a significant improvement in modeling convenience and productivity, not only comparing to repetitive graphical editing but also relatively to the PNML based specifications, which demand an exhaustive enumeration of all model elements. Even the enumeration in a simple textual language allows a much more compact and readable notation than XML.

In the second contribution, we propose a minimalist set of operations for model structuring and composition. These should be supported as part of the API.

The following section presents the UML class model for the APIs definition. After, we present two approaches for textual model specifications — through exhaustive enumeration and through model composition. This implies structural and label-based operations. Finally, we discuss related work and conclude.

2 A Model for Human-Friendly Textual Specifications for Petri Nets

In this section we present a list of requirements for readable and general textual languages for Petri net models specification. We argue that each of these languages should take the form of one framework for some general-purpose programming language, allowing their use as a domain language for Petri net modeling. The full power of those languages will still be available thus capitalizing on the user knowledge of their favorite programming language and, especially, on the availability of tools and documentation. Note that the interchange of models should still be guaranteed by PNML specifications, just like with other tools for the same or distinct Petri net classes. This was already presented in Figure 1.

The textual specification should seamlessly accommodate any kind of Petri net class as long as the respective model is specifiable using PNML. To that end, the API assumes the fundamental concepts already identified in the PNML Core Model (*vide* Fig. 2), namely the following assumptions:

1. All net classes should have places, transitions, and arcs;
2. Each Petri net model can have several pages;
3. Nets, places, transitions, and arcs can have associated labels, which allow the specification of the respective characteristics, according to the respective net class.

For the specification of the human-friendly APIs we propose the following requirements, which include the assumptions in the PNML Core Model presented before:

1. The API supports the following "top-level" objects, which are seen as first class objects in the used programming language, thus allowing the expected set of operations on them:

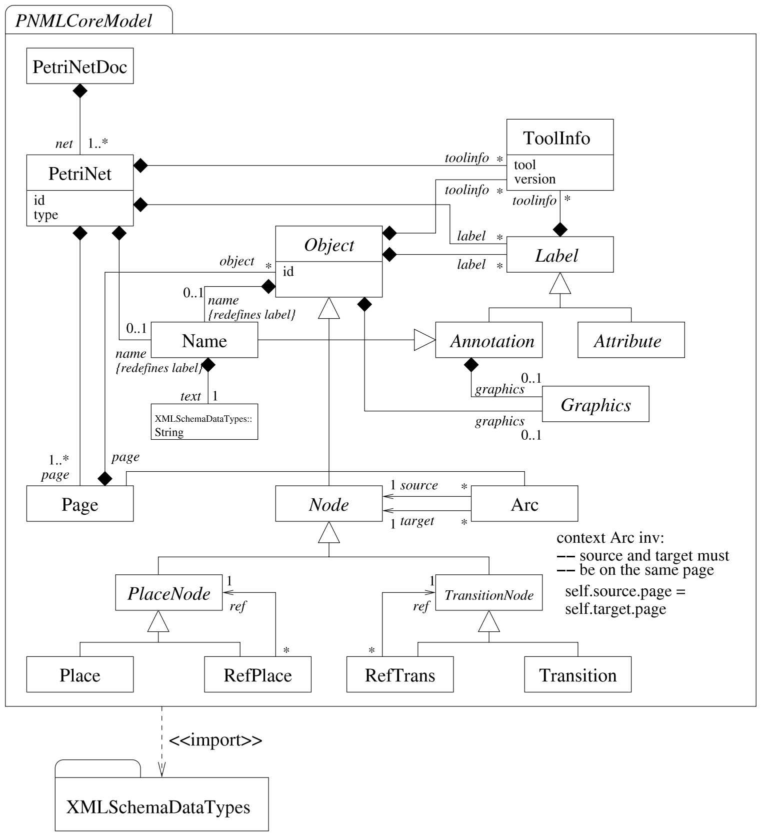


Fig. 2. PNML Core Model (*in* [2])

   - `NetClass`;
   - `Model` (supporting the `PetriNet` class in the PNML Core Model);
   - `Net` (supporting the `Page` class in the PNML Core Model);
   - `Place`;
   - `Transition`;
   - `Arc`.
2. The API allows a direct manipulation of net models, according to the PNML models' structure:
   (a) It is possible to read and assign values to each label of each `Net`, `Place`, `Transition`, and `Arc` object;
   (b) For each `Net` object, it is possible to define the respective set of places, transitions, and arcs, as well as the attribute values for each element;
   (c) It is possible to define the source and target for each arc in a compact and readable form.
3. Using the API it is possible to read, generate, and modify PNML models:
   (a) It is possible to create `Model` objects from PNML files;

   (b) It is possible to create PNML files from `Model` objects.
4. Each `NetClass` object corresponds to a Petri Net Type Definition (PNTD) [7].

These requirements have the objective of minimizing the gap between the metamodel and the model specification: the metamodel becomes closer to the objects effectively used by the modeller in the textual specification. Namely, PNTDs and PNML models can be specified in a human-friendly format: PNML models as programs; PNTDs as classes, inheriting from `NetClass`. Each of these classes is the specification, in the used programming language, for the specific Petri net type. More specifically, each class definition specifies the net, transition, place, and arcs labels, as well as the semantics for node fusion for each of those labels. This allows a full control over the necessary conditions for the nets to be composed: if there is no meaningful semantics for the fusion of some nodes depending on the node label values, or due to some other reason, then the class can explicitly forbid it, e.g. producing an error message and returning an error value.

The UML class diagram in Fig. 3 presents the proposed model supporting the human-friendly API, which we name *PnText Model*. Compared to the PNML Core Model, the PnText Model adds classes for all net element labels, namely `PlaceLabels`, `TransitionLabels`, `ArcLabels`, and `NetLabels`, and explicit lists for all net elements: `PlaceList`, `TransitionList`, and `ArcList`. Also, it supports the definition of new Petri net classes by extending the abstract class `NetClass`. It also replaces the composition relations, between models and nets and between nets and nodes, by an aggregation. While the PNML file clearly expresses a composition relations, the API should not enforce it as, e.g., a net (Page in the PNML Core Model) object and the respective node objects do not have coincidental life cycles: typically, when executed, the textual model (a program) creates a net, then the respective nodes and, possibly only after, the relation between the nodes and the net. Section 3.1 illustrates several of these object creation sequences.

3 Textual Specifications

As previously stated, the PnText model should be supported by a concrete textual language in the form of an API for a programming language. The diagram in Fig. 4 contextualizes the use of the textual models. Each concrete textual specification can be seen as the use of a domain language, and this can be implemented following one of two main approaches (e.g. [8]):

1. Defining a totally new language;
2. Adding a library to a well-known general-purpose programming language, thus allowing the specification of models as programs in the chosen programming language.

The first option is a laudable objective, yet a difficult one to achieve as it implies the need to completely define a new and *ad-hoc* programming language

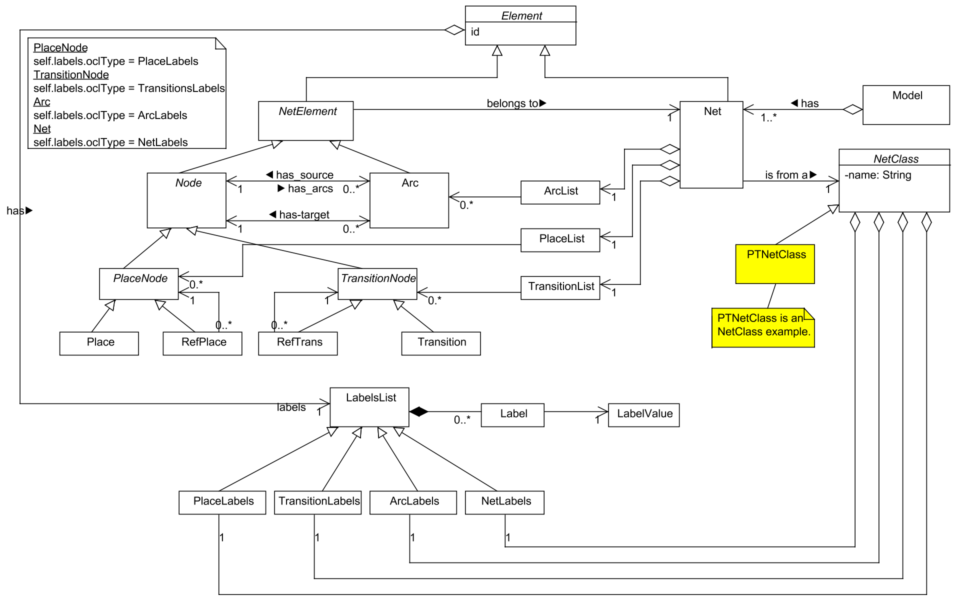


Fig. 3. Class diagram for the PnText library

if one wants to achieve the capabilities that are readily achievable through the use of a programming language as a domain language. It forces the modeller to learn a new language for which no, or few, tools will be available.

The second approach allows the full use of an already existing language thus taking advantage of all the available development environments. Yet, it may force some restrictions on the intended syntax.

The following section presents the proposed generic support for model specification, namely model definition by exhaustive enumeration (as in PNML, but in a more readable format), and model definition resulting from model composition.

3.1 Model Definition by Exhaustive Enumeration

Model construction by exhaustive enumeration is achieved through a direct use of the PnText model. Typically, the modeller, will define `Net` objects, and then add `Place`, `Transition` and `Arc` objects to them. The `Net` objects are added to the `Model` object. This type of model construction is a more human-friendly alternative to PNML specifications, which can also be conveniently generated from these textual models. For example, the following code shows a possible concrete syntax, using a general-purpose programming language (in this case Ruby [9]) and an API that implements the PnText Model, for the net `Producer` in Fig. 5 and writes it to a PNML file:

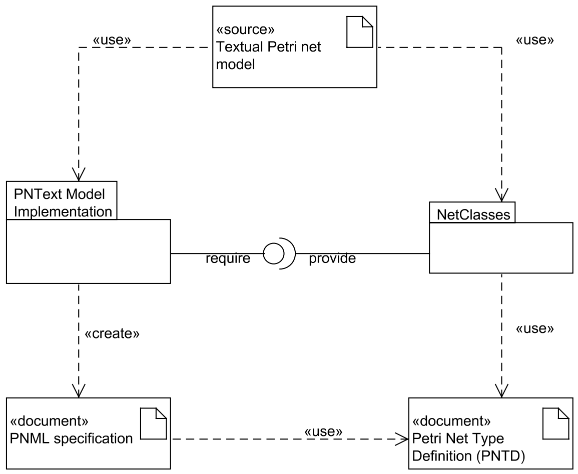


Fig. 4. System overview

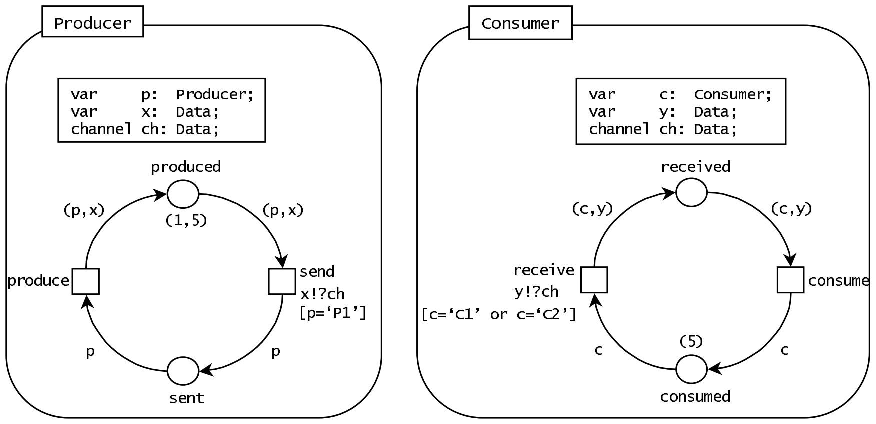


Fig. 5. Producer and Consumer nets with synchronous channel annotations (*vide* [10])

```
producer = Net(ColouredPetriNetClass, 'Producer',
               'var p: Producer; var x: Data; channel ch: Data;')
sentP     = Place('sent', '')
producedP = Place('produced', '(1,5)')
produceT  = Transition('produce', '', '')
sendT     = Transition('send', 'x!?ch', "[p='P1']")
producer.places      =  sentP | producedP
producer.transitions = produceT | sendT
producer.arcs = sentP     >> ['p']     >> produceT >> ['(p,x)']
                          >> producedP |
                producedP >> ['(p,x)'] >> sendT    >> ['p'] >> sentP
```

```
Model.nets = producer
Model.to_pnml('producer.pnml')
```

This specification defines objects that are created and added to the net as two operations. This allows the reuse of the objects definitions, e.g. the creation of new nodes based on copies of the existing ones. Yet, it also opens the door for incomplete specifications, as the modeller can forget to add the created nodes to the net. A way to avoid this kind of specification bug is to force the specification of the net when constructing each respective node. The following is a possible alternative, or complementary syntax:

```
producer = Net(ColouredPetriNetClass, 'Producer',
               'var p: Producer; var x: Data; channel ch: Data;')
sentP     = Place(producer, 'sent', '')
producedP = Place(producer, 'produced', '(1,5)')
produceT  = Transition(producer, 'produce', '', '')
sendT     = Transition(producer, 'send', 'x!?ch', "[p='P1']")
producer.arcs = sentP     >> ['p']     >> produceT >> ['(p,x)']
                          >> producedP |
                producedP >> ['(p,x)'] >> sendT    >> ['p']     >> sentP
Model.nets = producer
Model.to_pnml('producer.pnml')
```

As a final example, we present a syntax that is even closer to the PNML structure (again using Ruby):

```
producer = Net(ColouredPetriNetClass, 'Producer',
               'var p: Producer; var x: Data; channel ch: Data;')
sentP     = Place(producer, 'sent', '')
producedP = Place(producer, 'produced', '(1,5)')
produceT  = Transition(producer, 'produce', '', '')
sendT     = Transition(producer, 'send', 'x!?ch', "[p='P1']")
arc1      = Arc(sentP, produceT, 'p')
arc2      = Arc(produceT, producedP, '(p,x)')
arc3      = Arc(producedP, sendT, '(p,x)')
arc4      = Arc(sendT, sentP, 'p')
Model.nets = producer
Model.to_pnml('producer.pnml')
```

It is important to note that the full power of the underlying programming language is available to the modeller, as this provides an extremely versatile way to specify models. This is even more evident when we define models through the composition of other models. The following section presents a minimal framework allowing a textual specification for model composition for any class of Petri nets whose models are specifiable using PNML.

3.2 Model Definition by Composition

Most, if not all, mechanisms for Petri nets composition, structuring and abstraction have the objective to facilitate model construction (e.g. [11]). Hence, they all emphasize the need to construct models from submodels, the need to abstract models as nodes, or simply a hierarchical vision for large flat models. Petri nets already have the structuring mechanisms usually found in textual programming languages, which are desirable also in graphical specification languages. Yet, those mechanisms are defined *ad-hoc* by each tool for each Petri net type.

A common attitude is to identify model transformations that preserve properties. Here, we are not dealing with that problem. Instead, we propose a framework allowing the specification of model compositions for any kind of Petri net type whose models can be specified using PNML. The object is the pragmatics of specifying model construction for any Petri net type.

Based on an extensive literature, is clear that node fusion and node refinement provide the most intuitive and simple ways to structure and compose Petri net models, especially because they capitalize on Petri nets' graphical representation. Node fusion allows a graphical specification for asynchronous communication, using places, and also for synchronous communication, when using transitions. In fact, node fusion allows the modeller to take advantage of the Petri net graphical representation when structuring the model in a bottom-up approach. Node fusion is also the most obvious way to support top-down approaches to Petri net modeling: the well-known concepts of macrotransitions and macroplaces are clear examples where node fusion supports node refinement, definable as node removal followed by a set of node fusions between nodes in the supermodel and nodes in the submodel. An illustrative example can be found in Hierarchical Coloured Petri Nets [12]: the *substitution transitions* are defined as a set of place fusions between *socket places* in the superpage and *port places* in the subpage. Hence, we have chosen node fusion as the fundamental operation for model structuring. More specifically, we propose a structured way to use node fusion when composing Petri net models from any Petri net type.

Node fusion is a structural operation, thus we also have to specify how to fuse the respective annotations. Then, the annotations fusion can imply modifications in the respective arcs annotations and these can even imply changing the number and direction of each arc. Therefore, node fusion for arbitrary Petri net types brings the necessity of label fusion and arc fusion.

The result of these requirements is a small set of operations for the composition and modification of Petri net models. The operations are divided in two groups:

1. Structural operations;
2. Label operations.

The structural operations specify modifications in the model structure.

Label operations define how each kind of label should be transformed. They are specified as object methods for specific net classes, using the general-purpose programming language. To this part, we call the *fusion engine*, as it effectively specifies how the Petri net models, of a given Petri net type, are changed due to the node fusions in the structural operations. Note that these changes are different for each Petri net type, due to different annotations and semantics. Hence, for each Petri net type to be composed, we need to define a fusion engine. This is then implicitly used in the model. For example, when the modeller specifies a node fusion, the respective labels are transformed accordingly to the operations in the fusion engine definition (see the Label Operations section, below).

Structural Operations Node fusion is used for flat composition and also for hierarchical composition, as the basis for node refinement. Flat composition is supported by an operation named net addition that is defined as disjoint union followed by node fusion [13]. Hierarchical composition is supported by a refinement operation that is defined as node removal followed by net addition [14]. We also allow the simpler operation of node removal. We propose the following structural operations:

– $nodeRemoval : Net \times Node \rightarrow Net$ removes a single node from a net, together with all the arcs connected to itself, returning the resulting net;
– $nodeFusion : Net \times (Node)^* \rightarrow Net$ merges the nodes in the given nodes list and returns the resulting net;
– $disjointUnion : Net \times Net \rightarrow Net$ Returns two disjoint nets as a single net;
– $netAddition : nodeFusion \circ disjointUnion$ Node fusion after disjoint union.
– $refinement : netAddition \circ nodeRemoval$ Net addition after node removal.

The net addition and refinement operations allow bottom-up and top-down model construction, respectively.

To allow model modification for any Petri nets class we also have to deal with labels. This is presented in the following subsection.

Label Operations Although node fusion allows a clear syntax and semantics, this is only true for the graphical part. When dealing with labels, node fusion is as complex as the involved labels. Basically, we have to specify the operations that fuse the label values, or forbid node fusions when some specific label types are present. As these operations can be defined for any PNTD, they allow a unified view for all the well-known structuring mechanisms, whose definition is usually defined inside each tool.

The operations are defined at a low-level: for each label someone has to define how a new label value is generated based on the remaining label values of the fused nodes. In fact, that is one of the two main parts in the definition of the respective class (defined in the programming language). The other part is the specification of the labels (the names) for the element types (from PNML) net, place, transition, and arc. Finally, depending on the Petri net type, some label fusions, and hence some node fusions, may be forbidden or impossible, at least for some label values. These semantics can be defined in the label fusion operations.

Yet, for the modeler, the operations are not low-level. In fact, they are as high-level as they can be: after all, the modeler simply states which nodes get fused, just like it happens with existing tools, e.g. CPN Tools with socket/port places.

For each label in the respective PNTD, the class inheriting from the PnText model `NetClass` must define the following sets of operations:

- $\forall netLabel_i \in NetLabels, netLabel_i : fuseNetLabel^* \rightarrow NetLabel_i$ One operation is defined for each net label type; the operation receives all label values from a list of nets and returns a single label value of the respective label type;
- $\forall placeLabel_i \in PlaceLabels, fusePlaceLabel_i : placeLabel^* \rightarrow placeLabel_i$ One operation is defined for each place label type; the operation receives all label values from a list of places and returns a single label value of the respective label type;
- $\forall transitionLabel_i \in TransitionLabels, fuseTransitionLabel_i : transitionLabel^* \rightarrow transitionLabel_i$ One operation is defined for each transition label type; the operation receives all label values from a list of transitions and returns a single label value of the respective label type.

Node fusion can easily originate several arcs between the same pair of nodes. For this reason, the framework also allows the specification of an arc fusion operation between fused nodes, the same is to say, modifications in the arcs initially connected to fused nodes. This makes possible obvious simplifications, e.g., for place-transition nets, several arcs between a given place and transitions can be replaced by a single one having the sum of all weights as its weight. Note that this implies, at the minimum arc removal and, preferably, arc creation to create the new arc. For more complex Petri net types, less intuitive operations can be needed to simplify or correct the arc semantics for arcs that initially connect the fused nodes. In fact, they can even be forbidden or defined as an identity function thus avoiding any further modifications. For this reason, the operation to fuse arcs between fused nodes applies arcs and the respective labels into another list of arcs and respective labels, hence this label operation is in fact also a structural operation:

- $arcFusion : (Arc \times ArcLabels^*)^* \rightarrow (Arc \times ArcLabels^*)^*$ A list of arcs and the respective label values are applied into another list of arcs and the respective label values.

3.3 Net Addition

The first non-primitive structural operation is net addition. This is defined as a disjoint union followed by node fusion. The following concrete syntax (again in Ruby) exemplifies a possible textual notation for this operation, using the Producer and Consumer nets in Fig. 5, which we assume as already defined using the same programming language or created from PNML files:

```
result = producer + consumer ^
         (producer.t['send'] / consumer.t['receive'] >> 'sendReceive')
```

For the modeler convenience, the name label values are used. The identifiers (id attributes in the PNML specification) are automatically generated. Yet, this also means that, for each net, the names must be unique inside the respective set of places and the set of transitions. In the presented syntax, each net has an associative array for transitions (`t`) and another one for places (`p`). Note that the labels in places (named) `'send'` and `'receive'` need to be fused using a *fusePlaceLabel* operation. This operation, as well as the other label operations, are defined in the `ColouredPetriNetClass` which inherits from `NetClass` (see Fig. 3). The `ColouredPetriNetClass` defines the concrete operations (the *fusion engine*) for the corresponding PNTD.

The net addition operation allows bottom-up model construction. The following section, presents the refinement operation, which allows top-down modeling.

3.4 Refinement

The refinement structural operation is here illustrated for macrotransitions and macroplaces (e.g. [15]), the classical way to hierarchically structure Petri net models.

On the top-left corner, Fig. 6 shows a simple net using a macroplace node, which is an instance of the Subnet on the bottom-left. These two nets can be added resulting in the net on the right. Fig. 7 illustrates the same composition for a macrotransition. In both figures we use double bordered figures to represent the place and the transition that is refined, the same is to say, the one that is seen as a macroplace or macrotransition.

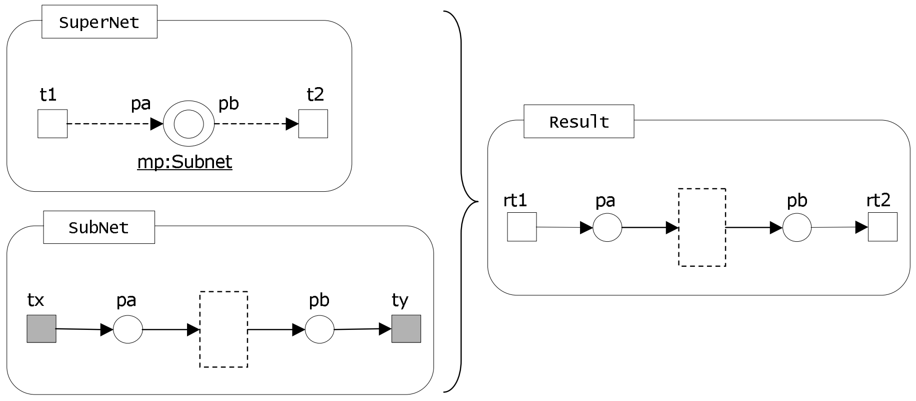


Fig. 6. A macroplace as net refinement

The presented macroplace interpretation is the following: the external transitions (in the supernet) put tokens in places inside the macroplace (e.g. place `pa`) and remove tokens from places inside the macroplace (e.g. place `pb`). The places in the subnet, connected to interface transitions (e.g. `tx` and `ty`), are "places that receive" (e.g. `pa`) or "places that offer" (e.g. `pb`).

The following specification is a possible concrete syntax (yet again in Ruby) for the net `Result` in Fig. 6:

```
result = superNet.clone()
result.remove(mp)
```

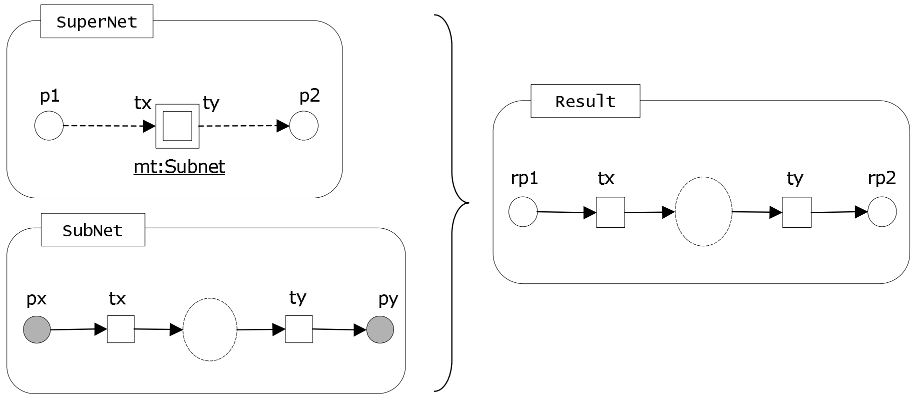


Fig. 7. A macrotransition as net refinement

```
result = result + subNet ^
            (result.t['t1']/subNet.t['tx'] >> 'rt1' |
             result.t['t2']/subNet.t['ty'] >> 'rt2')
```

The `result` is initialized as a copy of the `superNet`. After, the node to be refined is removed and the subnet is added by fusing the interface transitions.

The refinement operation allows a more compact specification, which is also more intuitively seen as top-down. The following is a possible concrete syntax:

```
result = superNet.mp >> subNet ^
            (superNet.t['t1']/subNet.t['tx'] >> 'rt1' |
             superNet.t['t2']/subNet.t['ty'] >> 'rt2')
```

For macrotransitions, the interpretation is similar: the transitions inside the subnet (e.g. `tx`) specify provided operations waiting for activation input parameters or that produce output (e.g. `ty`) and put them on receivers. The following, is a possible concrete syntax for the net `Result` in Fig. 7:

```
result = superNet.clone()
result.remove(mt)
result = result + subNet ^
            (result.p['p1']/subNet.p['px'] >> 'rp1' |
             result.p['p2']/subNet.p['py'] >> 'rp2')
```

Again, the refinement operation allows a more compact syntax:

```
result = superNet.mt >> subNet ^
            (superNet.p['p1']/subNet.p['px'] >> 'rp1' |
             superNet.p['p2']/subNet.p['py'] >> 'rp2')
```

In both cases, the refinement operation removes, and thus ignores, the arcs connected to the refined nodes in the supernets. This is coherent with the fact that if they are going to be refined these nodes should no longer be seen as such, but only as signaling abstractions in the supernet.

All the composition operators become much more powerful if the modeler is allowed to define several instances, copies, of a given `Net` object. The used programming language should provide the support for this.

4 Related Work

The use of textual languages for the specification of diagrams is obviously not new. In the last few years, several authors have pointed out this as a welcomed addition to the use of graphical languages, most notably UML (e.g. [16, 17]).

From the modeler perspective, the motivation for these approaches comes from the possibility of rapid prototyping and easy of use, especially when faced with the alternative use of large and complex graphical tools.

In the Petri net domain, numerous textual representations for net models exist, but most were designed to store models generated in graphical tools. Thus, their designs do not aim at human usability, as, typically, only tools will read and write specifications using these languages.

Other tools, especially model verification tools (e.g. [18]), allow the specification of models as text.

In the area of process algebras there are several significant proposals that provide solutions for the composition of Petri net models, most notably the Petri Box Calculus [19]. Yet, the motivation for the presented API is pragmatic: in industry and even in academia, it is much easier to find modelers and engineers who understand and can use some common programming language than the few who understand a process algebra. In fact, anecdotal evidence indicates that many engineers do not even know what a process algebra is. The presented set of operations are intuitive as they are based on the node fusion ubiquitous concept, which is used to define macrotransitions and macroplaces in a bottom-up or top-down fashion. It seems clear this is as simple as possible. The fusion concept is also used to handle the respective labels thus avoiding the need for further complexities. If, for a given net class, it is not possible to compose nets using these simple constructs, that is also expressed in the respective net class file. As already stated, that file specifies the PNTD defined labels for elements net, place, transition, and arc, as well as, the respective label fusion operations.

The PetriScript language [20] is the only language, we are aware of, that allows the specification of Petri net models in a human readable format and with the objective of providing the modeling capabilities of a programming language. It is part of the CPN-AMI set of tools [21]. As in the other cases, this textual language is designed for a specific class of Petri net. Besides, it follows the approach of defining a totally new language. This forces users to learn a new language with less available tools and constructs than popular general-purpose programming languages.

The PNML Framework [22] guarantees compliance with the standard but in the end provides a low-level interface for the programmer: e.g. a net with two places and two transitions demands a quite extensive textual specification, around 150 source code lines (including the graphical specification). Hence, the high level API implementing the PNText model, should use the PNML Framework to guarantee compliance with the standard, but should also provide a much more compact way to create and modify models.

The view of a net module as a source of instances is probably quite old and is also present, e.g., in Modular PNML [23]. In fact, net instances are just an example of the several ways in which the here proposed use of an underlying general-purpose programming language can bring additional benefits to the modeller, namely by providing extra structuring and abstractions mechanisms, e.g. vectors and other data structures, methods, templates, mixins, as well as data processing capabilities.

5 Conclusions

We have presented a model for the definition of textual specification languages for Petri net models based on the definition of APIs for general purpose programming languages. The use of a programming language also allows the specification of operations on the net, node, and arc labels. We have shown how this can be used to allow the specification of net models compositions, including bottom-up approaches based on net union and node fusions, and top-down approaches also using net union and node fusion. These node fusions imply the specification of label fusions, one for each net, place and transition. These are specified, for a given PNTD, as operations, inside a class, of the programming language. Also, the possibility of using instances, for each net, increases model compactness. This, together with the programming language constructs, clearly easies the creation of complex models in a compact, readable, and maintainable form.

We hope this paper can contribute to the creation of APIs for popular general purpose programming languages permitting their use as human-friendly textual languages for Petri net model specification, while allowing a direct translation to, and from, PNML models, in order to take advantage of already available tools.

Acknowledgements The authors thank the anonymous reviewers and the editor Ekkart Kindler whose insightful comments significantly helped us to improve the paper. This work is supported by the Portuguese FCT sponsored POSC/EIA/61364/2004 FORDESIGN project.

References

1. Barros, J., Gomes, L.: Towards a Human-Usable Textual Language for Petri Nets. `http://wwwcs.uni-paderborn.de/cs/kindler/events/PNS07/PDF/BarrosGomesPNS2007.pdf` (2007) Workshop on Petri Net Standards 2007, Satellite event of Petri Nets 2007, Siedlce, Poland.

2. ISO: ISO/JTC1/SC7/WG19. Software and Systems Engineering - High-level Petri Nets Part 2: Transfer Format. FCD 15909-2, v. 1.2.0, ISO/IEC (2007)
3. Society, S.F.: SDL-2000 new presentation – Drawings compared with text. `http://www.sdl-forum.org/sdl2000present/sld006.htm` (2007) (cited 2007-09-14).
4. Dijkstra, E.W.: On the Economy of doing Mathematics. In: Proceedings of the Second International Conference on Mathematics of Program Construction. Volume 669., London, UK, Springer-Verlag (1993) 2–10
5. Lakin, F.: comp.lang.visual Frequently-Asked Questions (FAQ). `http://www.faqs.org/faqs/visual-lang/faq/index.html` (1998) Answer to question 12: "Q12: What is the Deutsch Limit?". (cited 2007-09-14).
6. OMG: Unified Modeling Language: Superstructure. `http://www.omg.org/technology/documents/formal/uml.htm` (2007) version 2.1.1 formal/2007-02-05.
7. Weber, M.: Petri net type definition (PNTD) for a place/transition net. `http://www2.informatik.hu-berlin.de/top/pnml/pntd.html` (2003)
8. Fowler, M.: Domain Specific Languages. `http://martinfowler.com/bliki/DomainSpecificLanguage.html` (2004) (cited 2007-09-14).
9. Ruby Home Page: Ruby Home Page. `http://www.ruby-lang.org/en/` (2004)
10. Christensen, S., Hansen, N.D.: Coloured Petri Nets Extended with Channels for Synchronous Communication. Daimi PB-390 (1992) Also in: Valette, R.: Lecture Notes in Computer Science, Vol. 815; Application and Theory of Petri Nets 1994, Proceedings 15th International Conference, Zaragoza, Spain, pages 159-178. Springer-Verlag, 1994. Abridged version; available at `http://www.daimi.au.dk/CPnets/publ/full-papers/ChrHan1994.pdf`.
11. Gomes, L., Barros, J.P.: Structuring and Composability Issues in Petri Nets Modeling. IEEE Transactions on Industrial Informatics **1** (2005) 112–123
12. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use - Volume 1 Basic Concepts. 2^{nd} corrected printing edn. Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, Germany (1997)
13. Barros, J.P., Gomes, L.: Net Model Composition and Modification by Net Operations: a Pragmatic Approach. In: Proceedings of the 2th IEEE International Conference on Industrial Informatics (INDIN 2004). (2004)
14. Gomes, L., Barros, J.: On Structuring Mechanisms for Petri Nets Based System Design. In: Proceedings of the 2003 IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2003), IEEE Catalog Number: 03TH8696 (2003) 431–438
15. Silva, M.: Las Redes de Petri: en la Automática y la Informática. Editorial AC, Madrid (1985)
16. Fowler, M.: UmlSketchingTools. `http://martinfowler.com/bliki/UmlSketchingTools.html` (2004) (cited 2007-09-14).
17. Spinellis, D.: On the Declarative Specification of Models. IEEE Software **20** (2003) 94–96
18. Varpaaniemi et al., K.: PROD 3.4.01 An Advanced Tool for Efficient Reachability Analysis. `http://www.tcs.hut.fi/Software/prod/` (2007) All PROD authors are listed at `http://www.tcs.hut.fi/Software/prod/AUTHORS` (cited 2007-09-14).
19. Best, E., Devillers, R., Koutny, M., eds.: Petri net algebra. Springer-Verlag (2001)
20. Hamez, A., Renault, X.: PetriScript Reference Manual 1.0. `http://www-src.lip6.fr/logiciels/mars/DOWNLOADS/CPN-AMI/PetriScript_Reference_Manual.pdf` (no date) (cited 2007-09-14).
21. Laboratoire d'Informatique de Paris 6: CPN-AMI homepage. `http://www-src.lip6.fr/logiciels/mars/CPNAMI/index.html` (2006) (cited 2007-09-14).

22. Hillah, L., Kordon, F., Petrucci, L., Trves, N.: Model engineering on Petri Nets for ISO/IEC 15909-2: API Framework for Petri Net types metamodels. Petri Net Newsletter **69** (2005) 22–40
23. Kindler, E., Weber, M.: A Universal Module Concept for Petri nets. In: Proceedings des 8.Workshops Algorithmen und Werkzeuge fr Petrinetze / Gabriel Juhas und Robert Lorenz (Hrsg.) – Katholische Universitt Eichsttt, 2001. (2001) 7–12