PNML Framework: An Extendable Reference Implementation of the Petri Net Markup Language

L.M. Hillah[1], F. Kordon[1], L. Petrucci[2], and N. Trèves[3]

[1] Université P. & M. Curie - Paris 6, CNRS UMR 7606 - LIP6/MoVe
4, place Jussieu, F-75252 Paris CEDEX 05, France
Fabrice.Kordon@lip6.fr, Lom-Messan.Hillah@lip6.fr
[2] LIPN, CNRS UMR 7030, Université Paris XIII
99, avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France
Laure.Petrucci@lipn.univ-paris13.fr
[3] Cedric, CNAM, 292, rue St Martin, F-75141 Paris Cedex 03, France
nicolas.treves@cnam.fr

Abstract. The International Standard on Petri nets, ISO/IEC 15909, provides a formal semantics and syntax to enable model interchange and industrial dissemination. Part 2 defines a concrete interchange format as an XML-based language: PNML. This language is bound to evolve together with future developments of the standard.

This paper presents PNML Framework, a companion implementation of the standard. It provides developers of Petri net tools with a convenient and fast way to implement support of PNML documents. It abstracts away from any XML explicit manipulation and ensures compliance with the standard by using APIs.

Keywords: PNML, Petri nets standardisation, metamodels, MDE.

1 Introduction and Goals

The International Standard on Petri nets is divided in three parts. The first one deals with basic definitions of Place/Transition, Symmetric, and high-level nets.

The second part, ISO/IEC 15909-2, defines the interchange format for Petri net models: Petri Net Markup Language [8] (PNML, an XML-based representation). This part of the standard was published on November 11, 2009. It is now ready to be used by tool developers in the Petri Nets community.

Now, the standardisation group starts working on the third part. ISO/IEC 15909-3, aims at defining extensions and variations on the whole family of Petri nets. Extensions are for instance the support of modularity, time or probabilities. Variations consider less important semantic changes such as inhibitor arcs, bounded places etc. This raises the need to support such flexibility in the standard.

This paper presents PNML Framework: an API-based Framework to assist tool developers in achieving conformance with the standard. The motivations for PNML Framework are twofold:

– First, it provides tool developers with a programmer-friendly set of APIs which allows them to easily export/import compliant PNML documents. PNML Framework

J. Lilius and W. Penczek (Eds.): PETRI NETS 2010, LNCS 6128, pp. 318–327, 2010.

has been designed as a companion to the standard; it allows tool designers to manipulate Petri Net concepts instead of XML constructs and frees them from XML programming.
- Second, due to part 3, the standard is deemed to evolve and support different kinds of Petri nets. PNML Framework will provide a middleware software layer to cope with consistency of the required variations at the XML level.

The paper is structured as follows. Section 2 describes the Petri nets types metamodeling framework around which PNML Framework is conceptually built. Then it presents the architecture of PNML Framework and its use of the metamodels. Uses of the tool are presented afterwards. Section 2 ends by showing how Petri net tools can interact with PNML Framework. Section 3 reports a typical application example of model translation from PNML to COQ format. Finally, section 4 discusses how the design principles of the standard implemented by PNML Framework allows for flexibility and ability to evolve, strongly required for compatibility with the upcoming Part 3 of the standard.

2 PNML Framework Architecture and Services

As a companion to the standard, PNML Framework must support numerous kinds of Petri nets. So its design is based on a structured set of metamodels issued from the standard and describing components in the family of Petri nets.

This section quickly recalls the metamodels architecture that are detailed in [3]. For lack of space, we do not summarise the metamodels in this paper but [3] is available online. We only focus on the overall architecture of PNML Framework and the way this framework is intended to be used.

Metamodels for Petri nets. For the standard to be both robust and maintainable, the interchange format should convey structural information for Petri nets while being respectful of their semantic constraints. Thus, part 1 of the standard defines the semantics of several *Petri Net Types* (*i.e.* P/T, symmetric and high-level nets), while part 2 provides the associated metamodels.

Another challenge is the support of variations and extensions. To meet these issues, a metamodel-based approach as well as associated model engineering techniques, were chosen since they are tooled up and easily accessible. In addition they provide modular and incremental features to handle variations and extensions of these Petri nets types in an elegant manner, preserving their structural relationships.

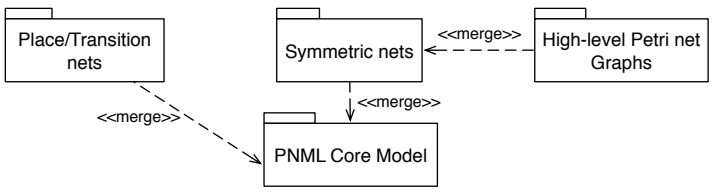

Fig. 1. Overview of the UML packages of PNML

The choice of model engineering techniques is driven by the state of the art of reliable approaches dealing with such issues. Although UML is a semi-formal modelling notation, its flexible levels of abstraction, expressivity, modularity and hierarchy make it appropriate for our goals. This is enforced by the fact that no semantical interpretation of Petri nets is required in an interchange format (only syntax is transferred).

So far, the standard provides a modular and incremental design of Petri net types metamodels. The metamodels of these Petri Nets types are encapsulated in UML packages. Fig. 1 shows their relationships, outlining the incremental design approach.

The *PNML Core Model* package (see Fig. 2) contains the basic structural definition of a Petri net as a labelled directed graph. All type specific information of the net is embedded in the labels. Labels are associated with nodes, arcs or the net. The *PNML Core Model* is intended to be the primary building block upon which concrete Petri net types are defined. Therefore, it imposes no restriction on labels because it is not a concrete Petri net type. For additional details concerning the metamodels, the reader is referred to [3].

As shown in Fig. 1, each concrete Petri net type is built either upon the *PNML Core Model* or upon another existing concrete Petri net type. The *Place/Transition Nets* package thus merges its definitions with the *PNML Core Model* ones, while the *High-level Petri Net Graphs* package merges its own with the *Symmetric Nets* ones. Each concrete Petri net type defines its legal labels by extending the primary definitions in the source package (*PNML Core Model* for P/T nets and *Symmetric Nets* for *High-level Petri Net Graphs*) and possibly adding some syntactic restrictions by means of OCL formulae (*e.g.* connectivity between places and transitions). The metamodels of these labels are

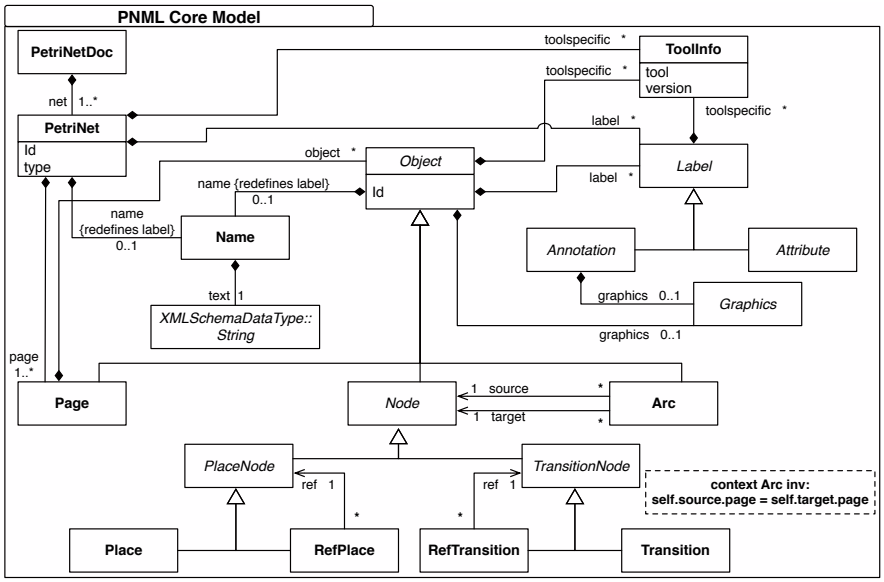

Fig. 2. Overview of the *PNML Core Model* package

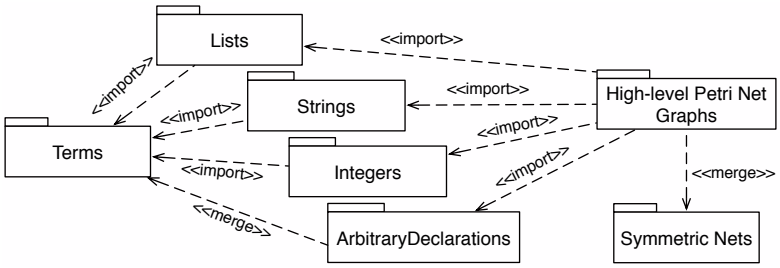

Fig. 3. *High-level Petri Net Graphs* reusing *Symmetric Nets* and importing type-specific labels

designed in specific packages so as to be reused as much as possible between related Petri net types.

Fig. 3 illustrates this extension mechanism from *Symmetric Nets* to *High-level Petri Net Graphs*. *High-level Petri Net Graphs* reuse *Symmetric Nets* definitions but extend their terms by adding new concepts: lists, strings, integers and arbitrary declarations. A more detailed presentation of the standardised Petri net types is provided in [3].

Architecture of PNML Framework. Components of PNML Framework are automatically generated from the metamodels in the standard, and thus reuse their structure. We chose to encode these metamodels using EMF [2] in Eclipse. EMF is one of the most advanced and mature model-driven engineering framework, as it supports UML, code generation and model transformation. These features are key characteristics for developing PNML Framework, as well as providing clean modelling facilities and generated APIs (*i.e.* a set of APIs) for tool developers. Therefore, EMF constitutes a suitable framework to rely on for constructing PNML Framework.

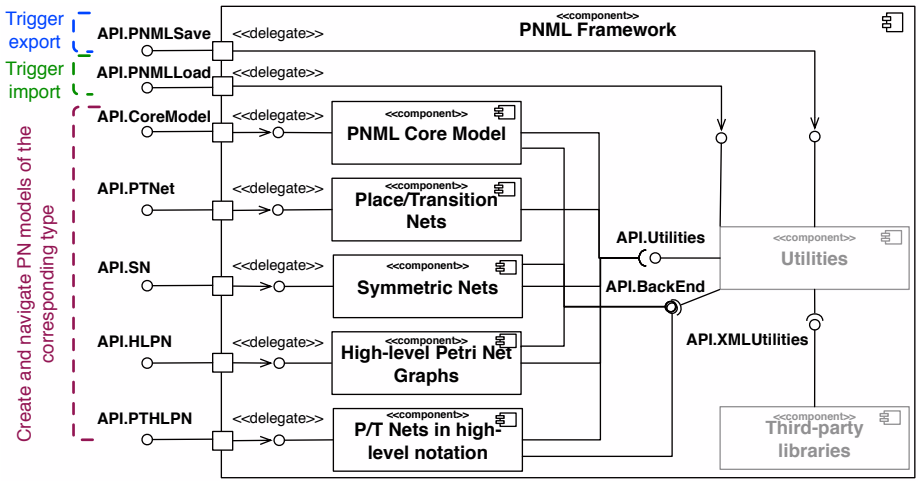

Fig. 4. PNML Framework architecture

Fig. 4 shows the structure of PNML Framework APIs. Each API (left part of the figure) manipulates a given Petri net type and is implemented by a dedicated component. The APIs are named like the corresponding piece of metamodel in the standard. The currently supported Petri net types are: *Place/Transition nets*, *Symmetric nets*, *High-level Petri nets* and *Place/Transition nets in high-level notation*, as defined in part 1 of the standard. Each component provides an API to be used to build models and navigate their structure.

There is no noticeable duplication between the components thanks to the technical UML *merge* operator (shown on Fig. 1) between the standardised Petri nets types in the metamodels. The *merge* operator includes the definitions of an existing Petri nets type into a new one. For instance *High-level Petri Net Graphs* merge *Symmetric Nets* in Fig. 3. As a result, every element previously in *Symmetric Nets* will then be included in the new type.

PNML Framework also embeds *Utilities* that tool developers can use to trigger the loading and storing of models into PNML documents. They can also use this component to turn on or off syntax validation for *PNML Document*. The *Utilities* component is responsible for loading PNML documents and figuring out what type of Petri nets they contain. It also sets up the export of Petri net models into PNML documents and their syntactic validation. This component also provides a workspace (or in-memory repository) where several Petri net models being handled can be stored. *Third-party libraries* are runtime components used by *Utilities*, providing basic APIs to manipulate XML trees. They are shown in grey in Fig. 4.

Most of the PNML Framework code is automatically generated. Manually developed code only concerns the *Utilities* component, which represents 3600 lines of code. This ensures maintainability in order to ease future developments resulting from part 3 of the standard. The next paragraph shows how PNML Framework can be used.

Uses of PNML Framework. Creating, for example, a place in any type of Petri net requires a single method call with the associated parameters such as name, marking and position (the method is automatically generated). Then, PNML Framework performs all the appropriate low-level EMF manipulations, in order to minimise the tool developers' efforts. The APIs primitives implement export and import of PNML elements:

- **Export**. Petri net models stored in memory, built as instances of Petri net types (w.r.t. their metamodels defined in the standard) are saved in a file compliant with the PNML syntax. PNML Framework takes care of the process, performing the required checks to produce the PNML document.
- **Import**. Petri net models are loaded in memory, from PNML documents, as instances of Petri net types. PNML compatibility checks are performed when loading models.

Fig. 5 illustrates the export and import mechanisms. It shows a Petri net model on the left-hand side as drawn by a tool user. A typical tool creates the object representation depicted in the centre of the figure. It can be exported by the appropriate API into a PNML file shown on the right-hand side. Importing PNML models is similar.

This process enjoys the independency between the tool internal representation and the current version of PNML. Compatibility concerns are handled by PNML Framework.

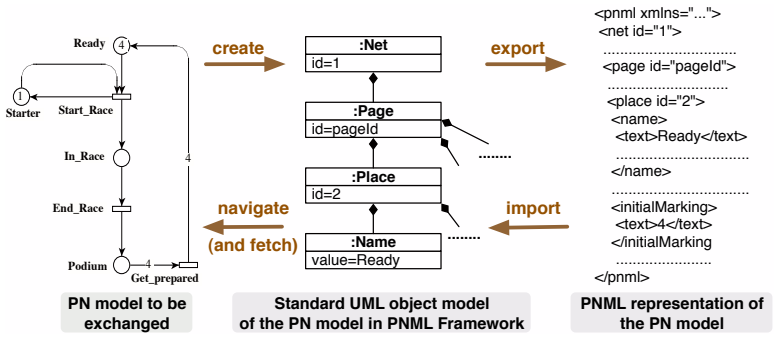

Fig. 5. Export and import of a Petri net model from PNML

Interaction schemes in using PNML Framework. PNML Framework is implemented in Java on top of Eclipse. It is also distributed as a standalone library. In order to use PNML Framework, a tool may be implemented in Java (not necessarily on top of Eclipse) or in any language supporting Java bindings. Fig. 6 depicts the ways to use PNML Framework in order to support the interchange standard.

Tool **T1** directly uses the provided API to export/import models in PNML. T1 is the typical example of a standard compliant Petri net tool that relies on PNML Framework. This is the case of Coloane [4].

Tool **T2** exemplifies the use of a standalone application that ensures the conversion of PNML files from/to an existing tool. This converter must parse/produce the T2 internal format. This is also the case of both the CPN-AMI [5] import/export facilities and the PNML2CoQ plug-in that is presented in section 3.

In contrast to T1 and T2, tool **T3** relies on its own implementation of the standard. Thus, support of PNML evolutions such as extensions and variations must be handled by T3 developers (with a risk of not conforming to the standard). The PNML web site [8] will be continuously maintained and provide updated versions of PNML.

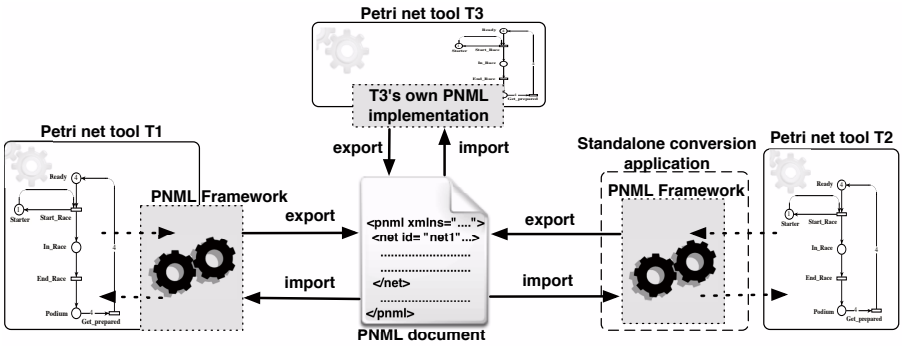

Fig. 6. Petri net tools exchanging PNML documents

```
ModelRepository.getInstance().createDocumentWorkspace("coqWksp");                1
PnmlImport pim = new PnmlImport();                                                2
pim.setFallUse(true);                                                             3
HLAPIRootClass imported = (HLAPIRootClass) pim.importFile("pnmlDocument");        4
Processor proc = MainProcessor.getProcessor(imported);                            5
    if (imported.getClass().equals(
    fr.lip6.move.pnml.ptnet.hlapi.PetriNetDocHLAPI.class))                        (5)
    { p = new PTProcessor(); return p;}
proc.process(imported, new PrintWriter(new FileWriter("coqDocument")));           6
```

Fig. 7. Source code of the PNML2COQ main function

3 Typical Use: PNML to COQ Example

This section illustrates the use of PNML Framework through the export of a Petri net in PNML format into a COQ theorem prover [7] representation as described in [1]. This example is representative of the situation of tool T2 in the previous section, even if it focuses on the import of models from PNML documents only (the export is very similar).

Overview of the process. The code snippets of Fig. 7 shows the key instructions to program the import of a *PNML Document*. There are six steps to complete the design of PNML2COQ (their numbers are shown on the right-hand side of the code):

1. create a workspace in PNML Framework where models can be manipulated (this is an in-memory repository to allow developers to work on several models during the same session),
2. create an *Importer* (from the *Utilities* package) to import a *PNML Document*, this document remains untyped at this stage (*i.e.* it can be any type of Petri net),
3. decide what to do in case the loaded Petri net type is unknown; *e.g.* downgrade to the closest type known by PNML Framework or to a specified one, or raise an error,
4. import the document (no work needed, this is provided by PNML Framework),
5. set the *processor* to be used for the loaded Petri net; this *processor* has to be written by the tool designer (as presented later in this section),
6. the *processor* is called to perform the desired operations (here, generate a COQ file).

Design of the Processor. A PNML document can contain several Petri nets, each of them composed of one or more pages. Hence, the *process* code (snippet on the left-hand side of Fig. 8) first handles nets (code on the right-hand side of Fig. 8) and then

```
public void process(HLAPIRootClass rcl, PrintWriter pw){
  PetriNetDocHLAPI root = (PetriNetDocHLAPI) rcl;
  for (PetriNetHLAPI net : root.getNetsHLAPI())
        processNets(net);
}
```

```
private void processNets(PetriNetHLAPI ptn) {
  //Some printout into the output Coq file...
  for (PageHLAPI page : ptn.getPagesHLAPI())
        processPages(page);
  //Some printout in the output Coq file...
}
```

Fig. 8. Code snippets from the processor

```
private void processPages(PageHLAPI page) {
  for (PageHLAPI pg : page.getObjects_PageHLAPI())
        processPages(pg);
  for (PlaceHLAPI pl : pth.getObjects_PlaceHLAPI())
        processPlace(pl);
  for (TransitionHLAPI tr : pth.getObjects_TransitionHLAPI())
        processTransition(tr);
  for (ArcHLAPI arc : pth.getObjects_ArcHLAPI())
        processArc(arc);
}
```

Fig. 9. Code snippet showing how Pages are handled

```
private void processPlace(PlaceHLAPI pla) {
  StringBuffer sb = new StringBuffer();
  nbplaces++;
  sb.append("Definition " + pla.getId() + " := mk" + "Place" + " " + nbplaces + ".");
  allPlaces = allPlaces + pla.getName().getText() +"::";
  sb.append("\n");
  sb.append("Definition m" + pla.getId() + " := (" + pla.getId() + ",0).");
  initMarking = initMarking + "m" + pla.getName().getText() +"::";
  print(sb.toString());
}
```

Fig. 10. Code snippet showing how places are handled

pages (snippet of Fig. 9). The processor uses a writer class to output the resulting Coq syntax into a Coq document (second argument in the *process* signature).

Fig. 9 details the processing of a page. It successively gets enclosed pages, places, transitions and arcs. All processing functions are written by the tool developer, according to his needs. This is eased by the iterators that PNML Framework provides for pages, places, transitions and arcs.

Fig. 10 shows how places are translated into Coq. It handles an object corresponding to a place, `pla`, accessing its attributes through the methods provided by PNML Framework (*e.g.* `pla.getId()`) so as to construct the output string in the Coq format.

All the examples in the figures above show that models are handled through the provided APIs only. Therefore, the tool developer using PNML Framework does not manipulate any PNML code. The processing we have exposed is fully implemented in PNML2Coq application. It can be reused for another export. In fact, PNML2Coq is inspired from the PNML2Dot tutorial available at [6].

The PNML2Coq application was implemented in one afternoon. The developer had no programming practice with Java but is an experienced programmer.

4 Achieving PNML Flexibility and Ability to Evolve

The standard is deemed to evolve (*i.e.* able to support new Petri net types introduced in part 3) and flexible (*i.e.* able to cope with additional information not in the standard). For both, PNML Framework guarantees standard compliance and thus preserves the ability of interchanging models with other tools.

Mechanisms for Ability to Evolve. The modular design of metamodels, as well as a compositional and incremental ways to build new Petri net types provide PNML with

the ability to evolve. This capability is crucial for the work on part 3 of the standard — addressing the definition of new Petri net types and structuring constructs.

Since PNML Framework has, from the start, been designed as a companion to the standard, its future enhancements will also follow the advances on the standard. This approach is both valuable for proof of concept purposes as well as future use by tool developers.

The work on addressing the mechanisms for an evolving standard should be fed by the Petri net community long-standing research achievements and recent results. We are therefore actively seeking theoretical as well as practical contributions from practitioners willing to share their new definitions and experiments.

Mechanisms for Flexibility. Moreover, flexibility of PNML allows tool developers to cope with tool-specific information in their models which is, of course, not included in the standard. For instance, if a tool associates C code with transitions, it can be introduced as *tool specific information* in the PNML document. PNML Framework supports this provision of the standard.

To do so, PNML Framework provides black box oriented PNML constructs, that allow any non-standard but well-formed XML constructions to be included in a PNML document. These can be included and retrieved by a tool-specific method (the non-PNML XML sentence is encapsulated within tool-specific tags). PNML Framework ensures consistent behaviours in import/export functions.

Thus, to embed some C code in a Petri net for instance, a tool must provide an XML representation of C programs. It might just be the whole C code embedded in a opening and closing XML tag, or a more elaborate syntax tree if the tool developer wants it to have that form.

Impact on the end-user. PNML Framework is maintained so as to be standard compliant and also to ensure backward compatibility with its former versions, starting from its current version 2.1, which implements the international standard (2009 version). This is possible thanks to the design choices.

So, if the metamodels evolve, the provided APIs will be regenerated so that the former are backward compatible with the new ones. The management of flexibility is orthogonal and thus not affected by evolution. So, maintenance is not impacted by standard evolution issues. Moreover, PNML Framework is designed to be upward compatible when new upgrades of the standard will appear (*e.g.* when introducing new extensions and/or variations in part 3 of the standard).

If tool developers want to implement their own Petri Net type or extend an existing one, they must provide the framework with its PNML-annotated metamodel, as well as its PNML grammar. Metamodels of the current Petri net types can be used as tutorials. Then, the code handling the new models is automatically generated. We have proceeded in this way to extend the P/T type with inhibitor, reset and read arcs.

5 Conclusion

PNML Framework has been designed as a companion and support of ISO/IEC-15909-2, which defines the PNML interchange format. PNML Framework provides a set of APIs

to read and write PNML files. This software is developed thanks to model engineering techniques (here EMF).

PNML Framework has been successfully used to quickly elaborate an export from Petri nets to COQ. The main design steps to build this application demonstrated the simple use of PNML Framework.

PNML Framework is open source and distributed under the Eclipse licence. It is implemented in Java. But as shown in this paper, import/export functions can be quickly developed as a standalone program for tools not being developed in Java.

PNML Framework enjoys flexibility capabilities and ability to evolve, which constitute a major issue in further development of the standard and free tool developers from maintenance issues due to its evolution. Initial successful experiments with small extensions such as inhibitor arcs etc. have assessed these objectives.

Acknowledgements. The authors are very grateful to Ekkart Kindler for his support and his comments on earlier versions of this paper.

References

1. Choppy, C., Mayero, M., Petrucci, L.: Experimenting formal proofs of Petri Nets refinements. In: Proc. Workshop REFINE (associated with FM2008), Turku, Finland, May 2008. Electronic Notes in Theor. Comp. Sci., vol. 214, pp. 231–254. Elsevier Science, Amsterdam (2008)
2. Eclipse Foundation. Eclipse Modeling Framework, http://www.eclipse.org/emf/
3. Hillah, L., Kindler, E., Kordon, F., Petrucci, L., Trèves, N.: A primer on the Petri Net Markup Language and ISO/IEC 15909-2. In: Petri Net Newsletter (originally presented at the 10th International workshop on Practical Use of Colored Petri Nets and the CPN Tools – CPN 2009), October 2009, vol. 76, pp. 9–28 (2009), http://www.cs.au.dk/CPnets/events/workshop09/assets/paper06.pdf
4. The Coloane home page (2009), http://coloane.lip6.fr/
5. The CPN-AMI home page (2009), http://www.lip6.fr/cpn-ami
6. The PNML Framework home page (2009), http://pnml.lip6.fr/
7. INRIA. The Coq Proof Assistant home page (2009), http://coq.inria.fr/
8. ISO/IEC/SC7/WG19. The Petri Net Markup Language home page (2009), http://www.pnml.org