ePNK: A generic PNML tool
Users' and Developers' Guide

Ekkart Kindler
Denmark's Technical University
DTU Informatics
DK-2800 Kgs. Lyngby
Denmark
`eki@imm.dtu.dk`

January 5th, 2011
version 0.9.0

Abstract

The *Petri Net Markup Language* (*PNML*) is an XML based interchange format for all kinds of Petri nets, which is currently under standardization [5]. Technically, PNML as defined in the the upcoming ISO/IEC 15909-2, is defining an interchange format only for high-level Petri nets and a simple version of Place/Transition systems. But, one of the objectives of PNML was to provide a means for exchanging any kind of Petri net [7, 14, 1]. To this end, the concept of a *Petri Net Type Definition* (*PNTD*) was introduced, which is subject of a newly issued standardisation project ISO/IEC 15909-3.

There are many tools supporting one form of PNML or the other and, in particular, there is the PNML Framework [6], which helps tool developers to ease the implementation of PNML by providing a framework and an API for loading and saving Petri net documents in PNML. This framework is based on the *Eclipse Modeling Framework* (*EMF*) [2] and has the focus on the underlying meta-models of Petri nets. The PNML Framework, however, is not generic in the following sense: Whenever a new Petri net type is created, the code for the complete tool needs to be regenerated. Moreover, the PNML Framework does not come with a graphical editor.

The ePNK overcomes these limitations: It provides an extension-point, so that new Petri net types can be plugged in to the existing tool without touching the code of the ePNK. For defining a new Petri net type, the developer, basically, needs to give a class diagram (actually an ecore-diagram) defining the concepts of the new Petri net type (and automatically generate some code from it), along with a mapping of these concepts to XML syntax. This type can then be plugged into the ePNK, and its graphical editor will be able to edit nets of this new type with all its features.

Actually, this was idea when we started the development of the *Petri Net Kernel* (*PNK*) about 15 years ago [11, 9, 12]. At that time, however, we had to implement all of the IDE functionality of such a tool ourselves. Today, we can make use of the eclipse platform [13], which helps us focusing on the Petri net specific parts; we get all the other functionality of a nice IDE, basically,

for free. This is why we named the tool *ePNK*: it can be considered to be an eclipse based Petri Net Kernel. But, it is just the spirit and idea that the PNK and the ePNK have in common; technically, there is not a single line of code from the PNK in the ePNK, and they are not compatible.

What is more, we use the nice features of EMF, GMF, and Xtext for developing the ePNK in a model-based way. In this way, the complete development process of the ePNK, is a case study in model-based software engineering using EMF and related technologies. This, actually, was the driving force behind this project. The evaluation and the lessons learned during this project, however, will be reported at an other occasion and to a different audience. This manual will focus on how to use the ePNK as an end user, and it will show how a developer can use the extension mechanisms of the ePNK for providing new Petri net types along with their XML syntax, and how to implement new functionality.

Contents

Chapter 1

Installation

This chapter discusses the installation of the ePNK. Readers who are just interested in getting an idea of what the ePNK is and who do not want to work with it right away, can skip this chapter.

1.1 Prerequisites

In order to install the ePNK, you need to have Java 1.6 (or higher[1]) and eclipse 3.5 (Galileo) installed on your computer.

For the installation of Java, please refer to `http://www.java.com/`.

Eclipse Galileo is not the most current version of eclipse. But you can still find it on the eclipse download pages at `http://www.eclipse.org/downloads/packages/release/galileo/r`. Eclipse is deployed in different configurations. If you are new to eclipse, it is recommended that you install the Eclipse Classic version. Download this version[2] and extract the file to some directory; after the extraction, you will find a folder named "eclipse" in this directory and in this folder, there will be an executable file also called "eclipse" (e. g. "eclipse.exe" on the Windows platform). Executing this file will start eclipse.

If you are new to eclipse, you can get a quick start to the Eclipse Integrated Develpoment Environment (IDE) at `http://www.vogella.de/`

[1]Actually, there seems to be a problem with the Oracle/Sun VM 1.6.0_21 on Windows and eclipse (see `http://wiki.eclipse.org/FAQ_How_do_I_run_Eclipse%3F`). You should use an older or a newer version. The notes on the bug-report suggest also that the eclipse versions 3.3-3.6 might not work together with Java 1.7 anymore.

[2]Downloading older versions of eclipse can take quite some time. Therefore, we made version of eclipse 3.5 for windows available for download at `http://www2.imm.dtu.dk/~eki/projects/ePNK/`.

`articles/Eclipse/article.html`. Once you have installed and started eclipse, you can also find much information in the “Workbench User Guide” in the eclipse help: You will get access to it in the “Help” menu in the eclipse toolbar under “Help Contents”.

1.2 Installing the ePNK in Eclipse

Once you have installed eclipse 3.5, you can install the ePNK from the eclipse workbench. To this end, the ePNK is made available as an eclipse update site at `http://www2.imm.dtu.dk/~eki/projects/ePNK/update/`.

To install the ePNK from there in your eclipse version, you should proceed as follows (after you have started it and selected a workspace):

1. In the eclipse tool bar, select “Help” $\rightarrow$ “Install New Software...”, which will open an install dialog.

2. In the install dialog, press the “Add...” button to add a new update site. In the “Add Site” dialog, enter some name (e. g. “ePNK Update Site”), enter the URL

   `http://www2.imm.dtu.dk/~eki/projects/ePNK/update/`

   as location, and then press okay.

3. Now, select the newly created ePNK update site in the still open install dialog. After some time, some ePNK items should pop up in the dialog. Select all of them and press okay.

   > **Note**: Make sure that the box “Contact all update sites during install to find required software” is checked; this will guarantee that all plugins from eclipse that ePNK needs, will be installed too (EMF, GMF, Xtext, etc.)

4. Follow through the installation process (don’t forget to accept the license agreement).

5. Then, the ePNK and all other required plugins will be installed; it is a good idea to restart eclipse after that (eclipse will ask you to do that anyway).

   Note: In version 0.8.0 of the ePNK, a dependency to the “EMF Validation Framework OCL Integration” feature was missing, so that this feature needs to be installed manually (at least if you want the full functionality of the ePNK). For version 0.8.0, you need to install the “EMF Validation Framework OCL Integration” separately: Select “Help” → “Install New Software...” another time; in the install dialog, select the standard Galileo update site, and de-select the option “Group items by groups” (otherwise, you won’t find this feature). Then select the “EMF Validation Framework OCL Integration” and follow through the installation process. From version 0.8.1 of the ePNK on, this dependency is made explicit, so that the “EMF Validation Framework OCL Integration” will be automatically installed when installing the ePNK from the ePNK update site.

Note: “Galileo” is not the the current release of eclipse! The ePNK, will ported to eclipse 3.6 (Helios) shortly.

Chapter 2

Introduction

This chapter gives a brief overview of the *Petri Net Markup Language* (*PNML*)) as well as on the concepts and ideas of the ePNK and its main features. In the end, of this chapter, there is some information for different kinds of readers on what to read and on how to read this manual.

2.1 Motivation

The PNML is an XML-based interchange format for all kinds of Petri nets, that allows different tools to exchange Petri net models among each other. One of its main features is that it is generic, which means that it provides a mechanism to define own types of Petri nets, which are called *Petri net type definitions* (*PNTD*). These Petri net type definitions define the additional concepts of the new Petri net type, as well as the representation of these new concepts in XML syntax. It is also possible that different tools include their *tool specific information* into PNML documents, which is information that can be ignored by other tools.

Up to now, there is no tool that fully supports these ideas, so that Petri net types, and tool specific extensions can be easily defined and plugged in to the tool and that comes with a generic editor, supporting all Petri net types, once they are plugged in.

The lack of this generic tool support was the starting point of developing the *ePNK*.

2.2 The Petri Net Markup Language

In order to better understand the ideas behind the ePNK, we discuss the main concepts and ideas of the PNML very briefly here. For more information on PNML and ISO/IEC 15909-2, we refer to [10, 5].

2.2.1 PNML core model

As stated above, the extensibility and genericity were two of the main objectives behind the PNML [8]. This is achieved by identifying the concepts that are common to all Petri nets in the so-called *PNML core model*. These common concepts are mainly the *places*, *transitions* and *arcs*, and that these arcs can have some kind of *label*. The PNML core model also takes into account that larger Petri nets can be split up into *pages* and connections between the nodes on the different pages can be established by *reference places* or *reference transitions*. And there are all kinds of graphical information that can be attached to the different elements, such as position, size, font-type, and font-size, etc.

In addition, the PNML core model defines the possible relation between these elements. In particular, it defines that places and transitions, which are generalized as *nodes*, are contained in pages and that arcs may connect these nodes. Figure 2.1 shows the PNML core model as an UML diagram.

Note that there is only one concrete type of label in the PNML core model, which is *name*. All the other possible labels need to be defined by a Petri net type definition, which will be discussed in Sec. 2.2.2.

In addition to the concepts and relations between them, the PNML core model also states some restrictions. For example, there is an OCL constraint stating that arcs can only be between nodes that are on the same page. But, there is no constraint yet that arcs can only run between a place and a transition or the other way round. The reason for that is that there are some kinds of Petri nets that would allow arcs between places or between transitions. This is why these restrictions would be part of a Petri net type definition.

Note also that the PNML core model does not specify concrete tool specific extensions. It is up to a tool to define what it needs. But, any tool must be able to read – and later write – any tool specific extension; their contents however, can be ignored.

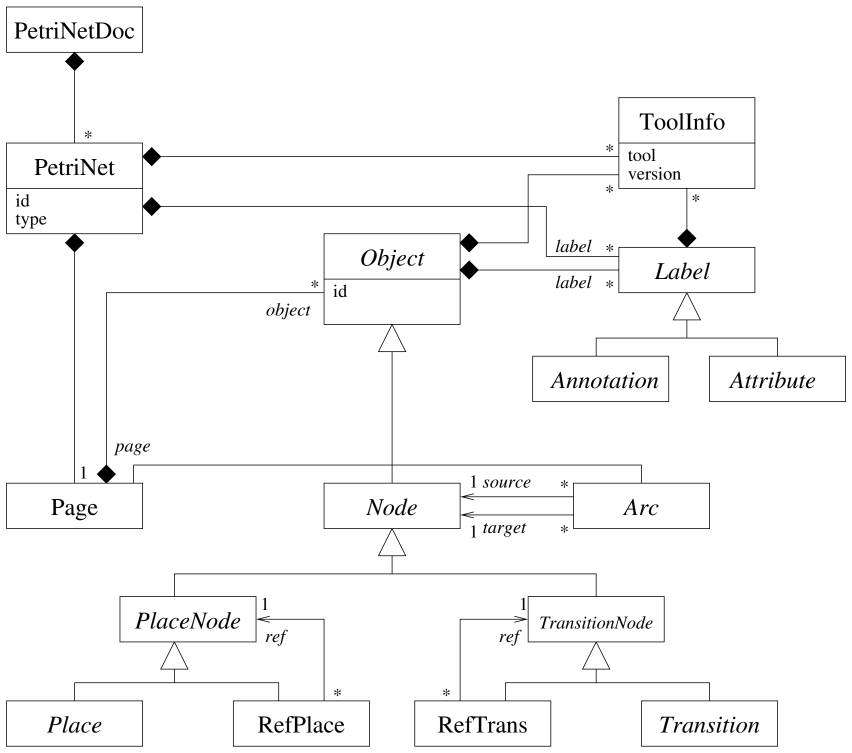


Figure 2.1: The PNML core model

2.2.2 Petri net type definitions

As stated above, it is the purpose of the Petri net type definitions to define which labels are possible in a specific kind of Petri net, and also to define some additional restrictions on the legal connections. Here we explain this idea by the help of a simple example: Place/Transition- Systems (P/T-Systems).

The two additional kinds of labels for Place/Transition-Systems are the initial marking for places, and the inscription for arcs. The initial marking can be any natural number (including 0) and the inscription for arcs can be any positive number. Figure 2.2 shows the UML model for these new concepts and how they are related to the concepts of the PNML core model.

In Fig. 2.2, there is also one additional OCL constraint. Without going into the details of OCL, this constraint states that an arc must run from a place to a transition or from a transition to a place. So, for P/T-Systems, it is no longer possible to connect places with places or transitions with

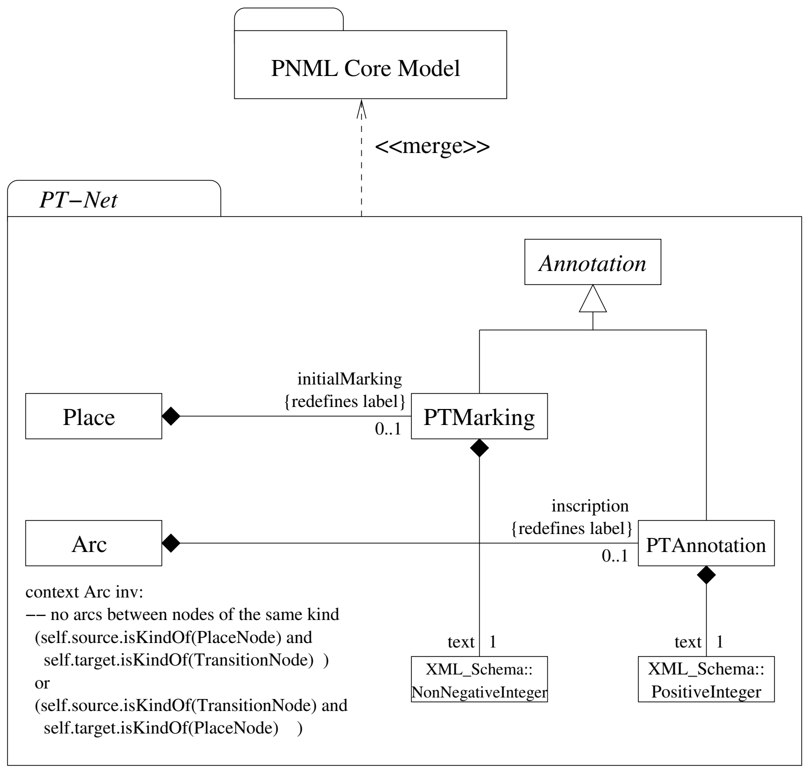


Figure 2.2: The PNTD for PT-Nets

transitions.

A Petri net type definition, would typically also define how the new concepts from Fig. 2.2 would be mapped to XML. But, for this simple kind of nets, this is actually done in a standard way.

2.2.3 Mapping to XML

As mentioned above, the PNML core model together with the model for a Petri net type definition, define the concepts of a specific kind of Petri net and how they can be connected. Therefore, these models are the centerpiece of PNML. Still, PNML is a XML transfer format for Petri nets. So, PNML also needs to define how these concepts will be saved or represented in XML. This is achieved by mapping every concept or feature in the models to some XML construct.

Here, we do not give these mappings, but rather show an example (for a

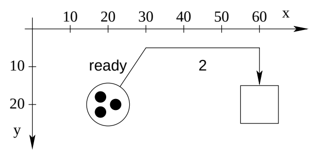


Figure 2.3: A simple P/T-System

detailed discussion of the mappings, see [5]). Figure 2.3 shows a simple example of a P/T-System and Listing 2.1 shows its representation in PNML's XML-syntax[1].

Note that this also shows an example of a tool specific extension: the positions of the individual tokens in the place.

2.3 ePNK: Objective

The main objective of the ePNK is to fully support the concepts of PNML, so that new Petri net types along with the mapping to XML syntax can be easily plugged into this tool – and to provide all the Petri net type definitions for supporting ISO/IEC 15909-2.

As soon as such a new Petri net type definition is plugged in, it should be possible to load and save nets of this type (and also documents containing net of different plugged-in types). And there should be a graphical editor that allows us to edit Petri nets of any plugged-in net type and is fully aware of all the features and the additional restrictions of the plugged-in net types.

For tool developers, the ePNK should provide an API to easily load and access Petri nets from PNML files, to manipulate them, and save them. Moreover, it should be easy to plug in new functionality for analysing and manipulating Petri nets.

2.4 How to read this manual

In this manual, we will explain the features of the ePNK in more detail.

This will cover the parts relevant for the "end user" who just wants to load, save and edit Petri nets of existing types and use some existing or plugged in functionality of the ePNK. In the rest of this manual, we will call "end users" just *users*. All the information for these users can be found in Chapter 3.

[1]We deleted some line-breaks to make this listing fit to single page

Listing 2.1: PNML code of the example net in Fig. 2.3

```
<pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml">
  <net id="n1" type="http://www.pnml.org/version-2009/grammar/ptnet">
   <page id="top-level">
    <name><text>An example P/T-net</text></name>
    <place id="p1">
      <graphics><position x="20" y="20"/></graphics>
      <name>
        <text>ready</text>
        <graphics>
          <offset x="0" y="-10"/>
        </graphics>
      </name>
      <initialMarking>
        <text>3</text>
        <toolspecific tool="org.pnml.tool" version="1.0">
          <tokengraphics>
            <tokenposition x="-2" y="-2" />
            <tokenposition x="2" y="0" />
            <tokenposition x="-2" y="2" />
          </tokengraphics>
        </toolspecific>
      </initialMarking>
    </place>
    <transition id="t1">
      <graphics>
        <position x="60" y="20"/>
      </graphics>
    </transition>
    <arc id="a1" source="p1" target="t1">
      <graphics>
        <position x="30" y="5"/>
        <position x="60" y="5"/>
      </graphics>
      <inscription>
        <text>2</text>
        <graphics>
          <offset x="0" y="5"/>
        </graphics>
      </inscription>
    </arc>
</page></net></pnml>
```

This manual will also cover the parts relevant for *developers*, who are interested in using the ePNK for their purposes and extending it by defining new Petri net types, by defining new tool specific extensions, or by new functionality[2]. Chapter 4 will provide the information for developers.

In addition, there will (eventually) be a part of this manual that discusses the architecture, the design, and some of the used technologies (and its problems). These parts might be interesting for developers, but actually addresses people interested in model-based software development technologies that are used (and extended) in this project: EMF, GMF, Xtext, ExtendedMetaData, EMF Validation, and OCL.

[2]Note that up to now, this needs to be done by the standard eclipse mechanisms. But, it is planned to eventually provide some functionality extension mechanisms for the ePNK that is tuned to Petri nets and allows to plug in ePNK functionality in a more uniform way.

Chapter 3

Users' guide

This chapter explains how to use the ePNK for creating, loading, saving, and editing Petri nets, and also how to use some of its functions. Since new Petri net types can be plugged in, we try to point out the general principles of these editors and how to use them. But for the particular syntax of some labels, it would be necessary to refer to the documentation of that specific extension. We will discuss these principles by some of the Petri net types that come with the basic version of the ePNK; and we use high-level nets (in terms of the ISO/IEC 15909-2 High-level Petri Net Graphs, or HLPNGs for short) to point out for which parts you would need to refer to the specific documentation of the specific Petri net type.

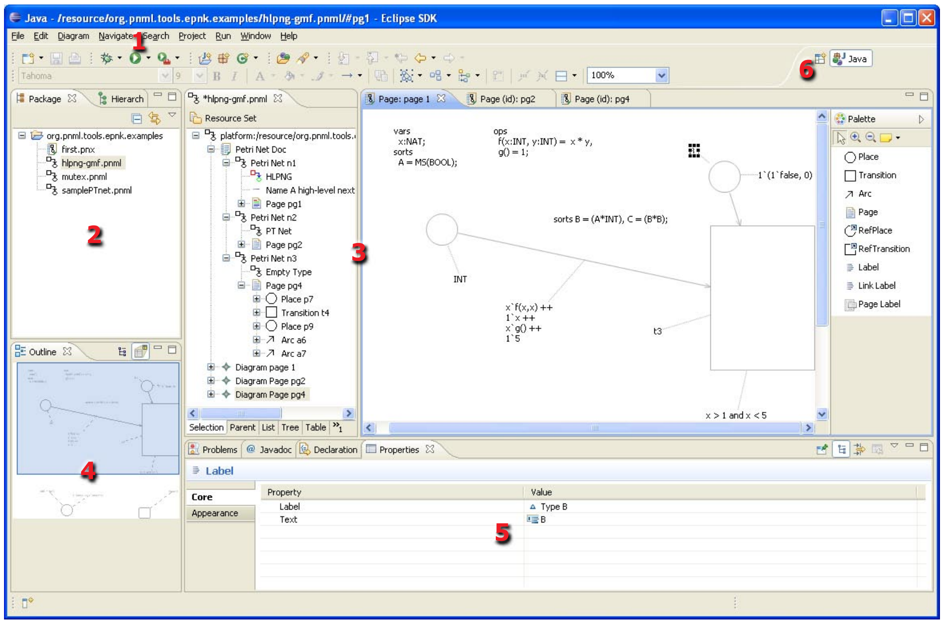
3.1 Eclipse as an IDE

For users who are new to eclipse and its IDE (Integrated Development Environment), we start with a brief overview of eclipse's workbench. Users who are familiar to eclipse already can directly read on in Sect. 3.2.

Once you installed and started eclipse (see Chapter 1), you will see the eclipse *workbench*. Depending on the chosen *perspective*, the different parts can be arranged in different ways. But, the principle behind is always the same. Figure 3.1 shows an example of the eclipse workbench, with some numbers marking some parts, which we will discuss next.

At the top (1), you can see the *menu bar* and the *tool bar*. Here, you will find the menus and tools for all the standard functionality, such as loading and saving files, and for standard editing operations. The menus that are shown in the menu bar depend on you installation and also on the editor that is currently active. For many operations, there are also the standard shortcuts, like CNTRL-S (on the windows platform) for saving the contents of an editor to a file. For getting more information, on that you could chose the "Help Contents" in the menu "Help" in the menu bar, and read the "Workbench User Guide".

> **Note:** In automatically generated editors, such as the graphical editor of the ePNK, the copy/paste functionality with CNTRL-C, CNTRL-V, CONTRL-X does not work properly without taking some extra measures, which we did not take yet. Therefore, you should not use the copy/paste shortcuts in these editors for now.



Property	Value
Label	Type B
Text	B

Figure 3.1: The eclipse workbench

On the left-hands side (2), you can see the *package explorer*, which gives you access to all the files in your workbench, and you can use it for browsing through the existing files and also to manipulate them, rename, copy, move, and delete them. This is very much like the file explorer of your operating system. Eclipse actually has different kinds of explorers, depending on the perspective and the user's preferences. The package explorer is made for java development projects. For our purposes, any of these would do, like for example the "navigator" or the simple "project explorer". To find and open one of these other explorers, you can use the menu “Show View” in the menu bar menu “Window”. All these explorers have some important concept in common, which concerns the organisation of files in the workbench: the top-level “folders” are actually not folders, but they are *projects*. This is only relevant when creating these projects. You can create a folder or file in the workbench only after you have created some project; this can be done via the “File” menu or by a right-click in a resource browser and then selecting “New”→ “Project”. Note that in the dialog, you can create many different kinds of projects; for us the project “Project” in category “General” will do. Then files can be created within this project.

In the center (3), you can see the *editor area* of eclipse. This is where all the editors started in eclipse will be opened. Note that there can be many editors open at the same time (in our example, there four editors open). Typically, you can only see one at a time and the others are hidden below it. But, you can move the editor tab to some border of the editor area, so that you can see the contents of two or more editors at the same time. In our example, there is a tree editor of a complete Petri net document open on the left-hand side, and on the right-hand side you can see one specific page open in a graphical editor (the graphical editors for some other pages are hidden beneath). Note that, even though there can be many editors open and even visible at the same time, there will always be only one editor that is active. This editor and what is selected there determines what you see in some other views. For example you can see the *overview* (4) of the page or you can see the property of the currently selected element in the *properties view* (5) at the bottom. In order to open an editor on some resource in the explorer, you would normally double click on the resource you want to open. This will open the *default editor* on the selected resource. You can use the right mouse button on a resource to open a pop-up menu and then select “Open with” to select a specific editor for this purpose. The way of how the content of a resource is edited very much depends on the kind of editor (mostly, it is straight forward). Saving the file can typically done by a shortcut (like CNTRL-S) in all editors (or via the “File” menu in the menu bar). An editor can be terminated (closed) either by clicking the close symbol on the tab of the editor or via the “File” menu in the menu bar.

Most editors support undo and redo of the latest changes, which you can access via the “Edit” menu or via the CNTRL-Z and CNTRL-Y shortcuts.

Note that the graphical editor for pages of a Petri net cannot be initiated directly from the explorer, since the resource could have many pages. The graphical editors for pages of a Petri net can be opened with a pop-up menu (right mouse button) on pages in the tree editor for Petri nets (see Sect. 3.4

for details).

All the other areas of the workbench are *views*[1]. In eclipse, views are used for many different purposes. The views that are most relevant for us, are the *outline* (4), the *properties* (5), and the *problems view* (not visible in Fig. 3.1). The outline gives an overview over the content of the currently active editor and in case of a graphical editor allows us to quickly move around in the visible area of this editor. The properties view shows some details of the currently selected element in the editor; in many cases, the properties view also allows us to edit some properties. Note that, initially, the properties view might not be open. You can, typically, open it from the active editor via a context menu on the right mouse button: In the pop-up menu that opens, there will be a menu "Show Properties View", which opens the properties view. You can also open the properties view via "Window"→"Show View"→"Others ..." and then selecting "Properties" in the category "General".

We mentioned already that the eclipse workbench can appear in different ways, which is defined by the chosen *perspective* (and some user-specific settings). The perspective can be changed via the tools at the top-right marked with (6), in our example. We do not need to change it; but if, for whatever reason, you end up in a wrong perspective, by clicking on the left symbol, you can open the "Open Perspective" dialog. There, you can select the perspective "Resource" or, if you like, "Java" (which is the default perspective).

If you are interested in more details in the eclipse workbench, you can have a look into the eclipse help ("Help"→"Help Contents") or at one of the many books or online articles; `http://www.vogella.de/articles/Eclipse/article.html` could be a start.

3.2 Creating Petri net files

This section explains how to create new ePNK files. Note that there are two ways ePNK can save Petri nets. The first and recommended way is PNML. The second is the XMI-serialisation of the PNML models, which we call PNX. Note that PNX, is part of the ePNK since XMI is the standard serialisation mechanism of the technology (EMF and ecore), and came for free. Whether PNX really should be a part of the ePNK distribution is yet to be seen. Therefore, the focus of this users' guide will be on PNML.

[1]Actually, also the resource browsers are views.

The easiest way of getting started with the ePNK is obtaining existing PNML files from somewhere else and just copy them to the workbench. For example, you could get some examples from the ePNK home page: `http://www2.imm.dtu.dk/~eki/projects/ePNK/`. You can also use create a simple text file with file extensions ".pnml" and insert the single line

```
<pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml"/>
```

to this file, which is an empty Petri net document without any nets in it.

The ePNK also provides you with a wizard for creating a PNML document. Like all eclipse creation wizards, this wizard will be started via the "New" menu, which can be either accessed by the "File" menu from the menu bar or via the popup menu that opens on a right-mouse click in the explorer. Then, select "Other..." (the short-cut to that would be pressing CNTRL-N in the explorer) and in the newly opened "Select a wizard" dialog chose "ePNK (PNML)" from the "ePNK" category and press "Next". In the next dialog, you must chose a name and, if you want, you can chose a different folder to which this file should be added. Pressing "Finish" will create the file and also open the tree editor on it (see Sect. 3.3); note that you also can continue the creation process by pressing "Next", which will allow you to chose an XML-Encoding. Note that in the dialog with the encoding, there is also a field asking for the "Model Object"; but you cannot chose anything here since PNML, in contrast to other formats, has a fixed root object that cannot be changed: "PetriNetDoc".

Note that in the same wizard category "ePNK" there is another wizard called "Pnmlcoremodel Model". When you use this wizard, a PNX file will be created. And in this wizard, you can select a root element different from the PetriNetDoc – but this would be reasonable only in very special cases (and when you know what you are doing).

3.3 The tree editor

As mentioned earlier already, the ePNK provides two kinds of editors for Petri nets. The *tree editor*, which allows us to create, modify, and delete all parts of the Petri net in a tree like structure. And there is a *graphical editor* in which a page of a Petri net with its places, transitions, and arcs can be edited in a graphical way. Clearly, the graphical editor is more convenient for editing pages than the tree editor. But, other parts like for example the page structure and the complete Petri net document are more convenient to edit in the tree editor. This is way we have both editors in the ePNK, and the tree editor provides access to the graphical editors.

3.3.1 The tree editor: overview

We have a closer look at the tree editors first. Figure 3.2 shows the eclipse workbench with two PNML documents open in tree editors. The right one shows the tree editor opened with the PNML file ("test.pnml") with the single line as discussed in Sect. 3.2. Therefore, it contains only the Petri net document element without any contents. The other PNML document open on the left-hand side ("hlpng-gmf.pnml") shows a Petri net document with three nets that have different types.

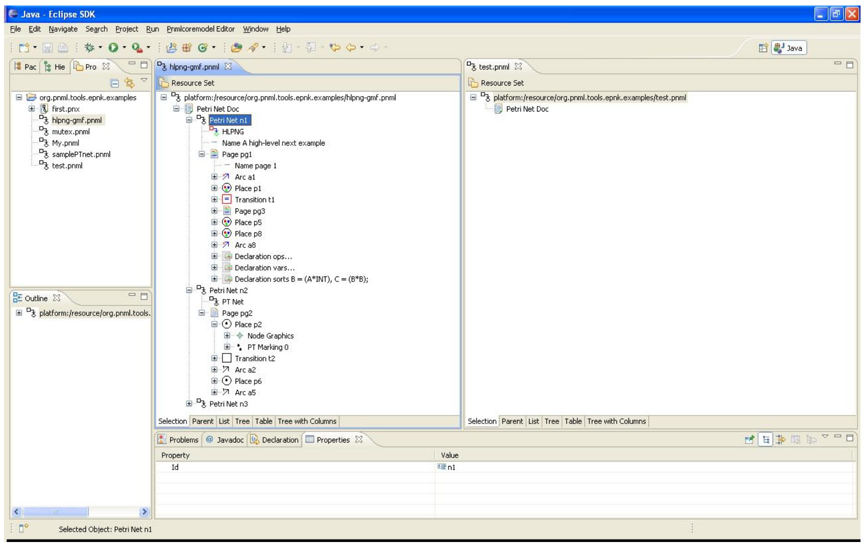

Java - Eclipse SDK
File Edit Navigate Search Project Run Pnmlcoremodel Editor Window Help
Java
Pac Hie Pro
org.pnml.tools.epnk.examples
first.pnx
hlpng-gmf.pnml
mutex.pnml
My.pnml
samplePTnet.pnml
test.pnml
Outline
platform:/resource/org.pnml.tools.
hlpng-gmf.pnml
Resource Set
platform:/resource/org.pnml.tools.epnk.examples/hlpng-gmf.pnml
Petri Net Doc
Petri Net n1
HLPNG
Name A high-level next example
Page pg1
Name page 1
Arc a1
Place p1
Transition t1
Page pg3
Place p5
Place p8
Arc a8
Declaration ops...
Declaration vars...
Declaration sorts B = (A*INT), C = (B*B);
Petri Net n2
PT Net
Page pg2
Place p2
Node Graphics
PT Marking 0
Transition t2
Arc a2
Place p6
Arc a5
Petri Net n3
Selection Parent List Tree Table Tree with Columns
test.pnml
Resource Set
platform:/resource/org.pnml.tools.epnk.examples/test.pnml
Petri Net Doc
Selection Parent List Tree Table Tree with Columns
Problems Javadoc Declaration Properties
Property
Value
Id
n1
Selected Object: Petri Net n1

Figure 3.2: Two Petri net documents in tree editors

These documents were opened from the explorer by a double click[2]on the respective file in the workbench explorer. Let us briefly go through what you see in the Petri net document "hlpng-gmf.pnml". The top-line shows the actual *resource* or file in which this document is stored; the second line is the symbol for the Petri net document itself – all documents will follow this structure in the tree editor. Then you can see that there are three Petri nets contained in this document (with *ids* n1, n2, and n3); the last one is actually not folded out because its was not fitting to the screen. The first line below the Petri net is the type of the Petri net. The first net is a high-level net,

[2]Remember, that you can use the pop-up menu to make an explicit choice by which editor you want to open the file. This way, you can open the file with a text editor, so that you can see the PNML it produces. On a double click, the file will always be opened with the editor you had selected last time.

which is named HLPNG according to ISO/IEC 15909-2; the second net is a Place/Transition-System, called PTNet according to the standard. You can also see that the nets contain places, transitions and arc and also some pages and sub-pages, which are indicated by corresponding icons. Note that these icons are different for the two different Petri net types, so that it easier to see what they are visually. In the properties view at the bottom, you can see the properties of the currently selected element, which is Petri net n1, and the only property is its id. Note that this net also has a name; this, however, is not shown as a property, but as a *child element* of the net, which is true for all labels of Petri nets. In this case, the name is "A high-level next example".

3.3.2 Creating elements

You can unfold all the sub nodes (children) of the net and this way inspect the complete document in all details. More importantly, however, you can create the basic elements of the Petri net document. You can create new nets, their type, and their pages. And from there, you would use the graphical editor to draw the rest. This basically works by inserting *child elements*. This is done by right-clicking the element to which we want to add a child, then selecting "New Child" in the dialog that pops up and then selecting the appropriate element. Figure 3.3 shows the pop-up dialog when inserting a new Petri net to the Petri net document.

The type of a Petri net, its name, and its pages can be inserted in the same way. Note that it is important that every Petri net is assigned a type right after it was created; and after its creation, the type should never be changed again. Otherwise, it could happen that a Petri net contains elements that do not make sense in a specific type. The tree editor does not enforce this yet[3]. There will be some more information on the Petri net types that are deployed as part of the ePNK in Sect. 3.5.

3.3.3 Saving the document

As discussed in Sect. 3.1, you can save the net via the "File" menu or with the CNTRL-S shortcut. Note that saving the net should always be done in the tree-editor, for now; saving from the graphical editors works, but the

[3]Note that it would be easy to enforce this, and we might eventually implement this. In the current experimentation phase, we leave that in as a feature to play around an experiment with this. But, you should really now what you are doing, when changing the type later.

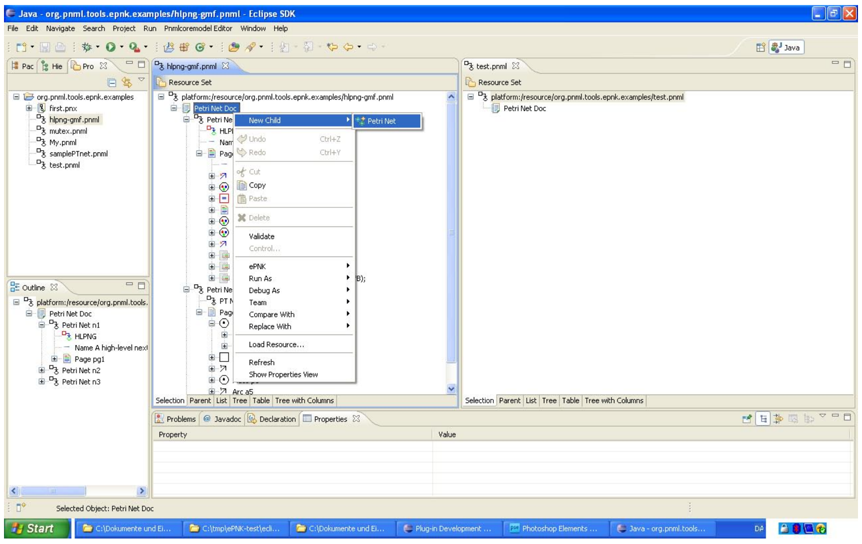


Figure 3.3: Pop-up menu when inserting a new Petri net

dirty bit is not reliable. So, in after finishing the editing a net, close all the graphical editors and explicitly save the net from the tree editor again.

3.3.4 Validating and correcting the document

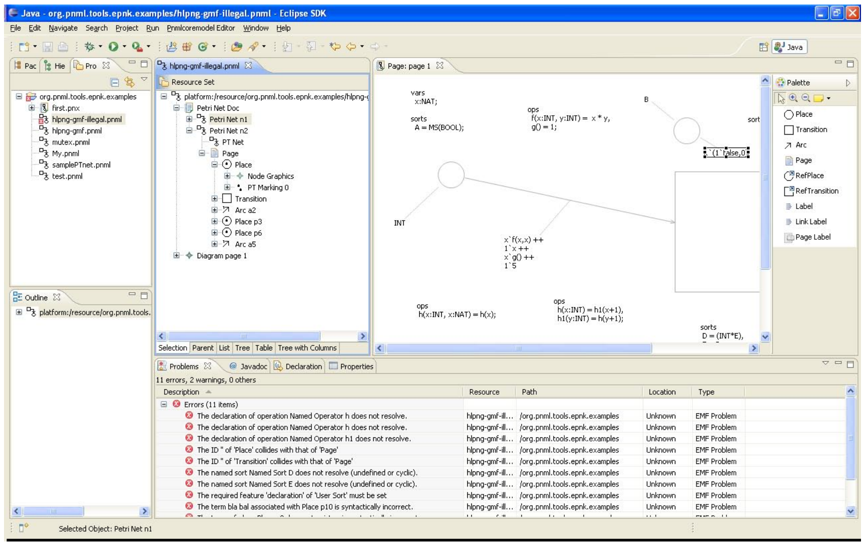
Before saving the document, it would be a good idea to validate it, which checks whether all the constraints that PNML imposes on Petri nets are met. It is possible to save a document that does not properly validate, and you would be able to load the file again. But, if you save a file that does not properly validate, you cannot be sure that the saved document is ISO/IEC 15909-2 conformant PNML.

There are many things that could be wrong and need validation on a Petri net document. Most of them are type specific (such as requiring that an arc must run from places to transitions or the other way round only); these Petri net type specific constraints will be discussed in Sect. 3.5. But, there are also some general constraints:

- Every Petri net object must have an id and this id must be unique in the scope of this document.
- Arcs may only connect nodes which are on the same page (as long as you are using the graphical editor, this constraint cannot be violated; but if you do changes in the tree editor, this could be violated).

- There must not be cycles on the references between reference nodes, and all reference nodes must refer to a node.
- A reference node must refer to a node within the same net.

In order to identify which constraints are violated, you can use the validation feature. Click the right mouse button on the Petri net document and, in the pop-up menu, select “Validate”. The result will be shown in a dialog; and the results will be also visible in the *problem view* later as shown in Fig. 3.4. Most of these errors are actually coming from high-level nets. But, there is also a general constraint violated in this example: some IDs collide (line 5 and 6 in the problems view). Remember that, if for some reason, the problems view is not open in your workbench, you can open it with “Windows”→“Show View”→“Problems”.


Java - org.pnml.tools.epnk.examples/hlpng-gmf-illegal.pnml - Eclipse SDK
File Edit Navigate Search Project Run Pnmlcoremodel Editor Window Help
hlpng-gmf-illegal.pnml
Page: page 1
Palette
Place
Transition
Arc
Page
RefPlace
RefTransition
Label
Link Label
Page Label
Selection Parent List Tree Table Tree with Columns
Problems Javadoc Declaration Properties
Selected Object: Petri Net n1

11 errors, 2 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (11 items)				
The declaration of operation Named Operator h does not resolve.	hlpng-gmf-ill...	/org.pnml.tools.epnk.examples	Unknown	EMF Problem
The declaration of operation Named Operator h does not resolve.	hlpng-gmf-ill...	/org.pnml.tools.epnk.examples	Unknown	EMF Problem
The declaration of operation Named Operator h1 does not resolve.	hlpng-gmf-ill...	/org.pnml.tools.epnk.examples	Unknown	EMF Problem
The ID '' of 'Place' collides with that of 'Page'	hlpng-gmf-ill...	/org.pnml.tools.epnk.examples	Unknown	EMF Problem
The ID '' of 'Transition' collides with that of 'Page'	hlpng-gmf-ill...	/org.pnml.tools.epnk.examples	Unknown	EMF Problem
The named sort Named Sort D does not resolve (undefined or cyclic).	hlpng-gmf-ill...	/org.pnml.tools.epnk.examples	Unknown	EMF Problem
The named sort Named Sort E does not resolve (undefined or cyclic).	hlpng-gmf-ill...	/org.pnml.tools.epnk.examples	Unknown	EMF Problem
The required feature 'declaration' of 'User Sort' must be set	hlpng-gmf-ill...	/org.pnml.tools.epnk.examples	Unknown	EMF Problem
The term bla bal associated with Place p10 is syntactically incorrect.	hlpng-gmf-ill...	/org.pnml.tools.epnk.examples	Unknown	EMF Problem

Figure 3.4: The problems view with many constraint violations

In order to reduce number of errors, you can also do a validation of sub-elements of the Petri net document, which could be a net or even a single page. Ultimately, however, you must validate the complete Petri net document.

In our example, there are some problems that could be fixed automatically. For example, unique ids could be set automatically. The ePNK provides an action for this. To this end, select the Petri net document, click the right mouse-button, and then select “ePNK”→“Add missing IDs” in the pop-up menu. This will fix all problems with the ids within a Petri net document.

There is another tool, which will fix some of the type specific problems. In high-level nets, many of the labels of the net are related to each other. For example, there are variable, function, and sort declarations on pages. And these variables, functions, and sorts could be used at other places, i. e. in other labels. To establish the connection between the use and the declaration of these symbols, the labels need to be linked with each other. Since this is part of the Petri net type definition interface, this can be done in a generic way. To properly link the *symbols* of any such type, you can select the Petri net in the tree editor, click the right mouse button, and select "ePNK"→"Link Labels (globally)". We will discuss more details in Sect. 3.5.

3.3.5 Other Petri net information

In principle, you can inspect and edit all the information of a Petri net document in the tree editor. In our example from Fig. 3.3, you can also see some labels (declarations of high-level nets in this case, or a marking of a P/T-System) or graphical information. If you have a closer look at these examples, you will also find some other types of elements such as tool specific information – and once graphical editors are started some auxiliary data. But, it is strongly recommended not to change any of this information in the tree editor[4].

3.4 The graphical editor

For editing the contents of pages, the graphical editor should be used. This graphical editor can be opened by right-clicking on the respective page in the tree-editor and, in the pop-up menu, selecting "ePNK"→"Start GMF Editor on Page".

Figure 3.4 shows the graphical editor with an opened page from a high-level Petri net (in this case one with several errors in it). Normally, this new editor shows on top of the tree editor, but it can be moved to the right side (click in the tab at the top of the editor window and move it while keeping the mouse pressed), so that the tree editor as well as the graphical editor are visible at the same time.

[4]Once the features of the ePNK that are really needed are fixed, the parts that should not be edited in the tree editor will probably removed or at least made read only.

3.4.1 Overview of the graphical editor

On the left-hand side of the graphical editor for the page, you see the *canvas* with all the Petri net objects on that page represented in a graphical way. This includes also the *labels*, which are either attached to an object by a dashed line or attached to the page itself, in which case it is called a *page label*.

At the top, you see the *tab* of this page, which also shows the page name (if the page has a name label assigned) or its id, or the path to this page (if the page has neither a name nor an id).

On the right-hand side, you see the *palette* or tool bar of the graphical editor. These tools allow you to create all the Petri net objects. Note that you can also create sub pages.

There are two different tools for labels. The tool “Label” is for creating labels that are attached to objects, the tool “Page label” is for creating labels that are directly attached to the page that is shown in this editor.

For creating objects and labels, you first select the tool by clicking on it, and then clicking somewhere into the canvas. For creating an arc, you select the arc tool and then click on the source object, and keeping the mouse pressed and move the mouse to the target object. Note that the arc is not added between two objects, if the Petri net type you are editing does not allow this.

3.4.2 Labels

When creating a new page label on a page, the graphical editor will show you all the possible options of legal labels for that net via a pop-up menu. Figure 3.5 shows the pop-up menu during the creation of a page label for high-level nets, where the only option “Declaration” is shown here. You can select an option, after which a label of that kind will be created. You can also abort by either pressing the “ESC” button or clicking somewhere outside the menu.

The process for creating a label is a bit different. First you can create a label, which however will not be attached to any object yet. This will be indicated by the text “<not connected label>”. Such a label can be connected to some Petri net object (also to a sub-page) by choosing the tool “Link Label”, clicking on an object, and without releasing the mouse button moving it over the not yet connected label. Then, a pop-up menu will be opened showing you the possible options of labels for the chosen object. This is shown in Figure 3.6 for a label that is attached to a place. The possible

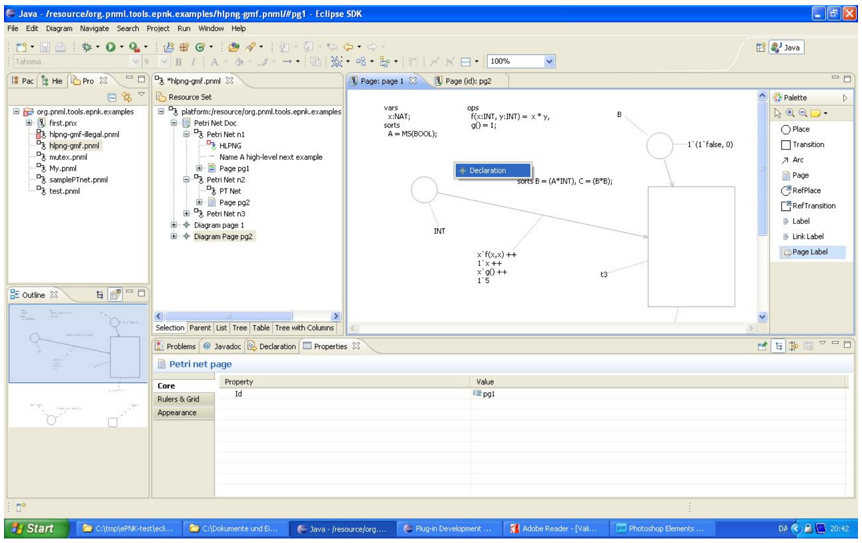


Figure 3.5: Pop-up menu during page label creation

options are "Name" (which is a legal option for any object, but only if there is no name attached yet) or "PTMarking 0", which is the initial marking for P/T-Systems (where "0" is the default value). After the selection, the label of the chosen type will be attached to the object. Again, attaching the label can be aborted by pressing "ESC" or by clicking somewhere outside the pop-up menu.

After a label was attached to an object it can be edited "in place", by clicking into it and pressing the ENTER key in the end. The legal syntax of the label depends on the Perti net type and which kind of label it is. In general, editing of labels as well as of page labels can be distinguished into two cases. The first case are *simple labels*, which typically are numbers or simple values like "true" or "false". If such a label is typed in syntactically incorrectly, the new value will be rejected, and the value of that label will be reverted to the value it had before editing. The other case, are *structural labels*. These are typically, labels with a complex syntax, as for example the declarations of a high-level net (actually all labels of high-level nets except the names are structural). These labels will also be parsed and checked for syntactical correctness; but the entered text will be stored in all cases. If the text is incorrect, however, the structure is not set and this will very likely result in some validation error later (see Sect. 3.3.4). So this error needs to be fixed, by editing the label again. In case of such an error, the text of

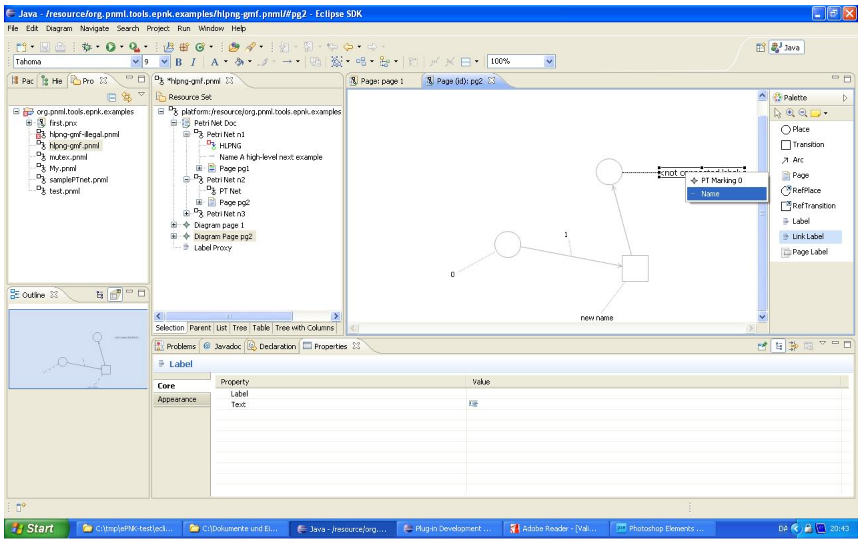


Figure 3.6: Pop-up menu during attaching a unconnected label to an object

the label will be shown between "`<!` " and " `!>`". If for example, we delete the comma that separates the two sort declarations in the label "sorts B = (A*INT), C = (B*B);" this will be shown as "`<!` sorts B = (A*INT) C = (B*B); `!>`". And upon validation, a validation error message will be given and shown in the problems view.

The documentation of the legal syntax of all type specific labels, in particular the one of the structural labels, is part of the documentation of the Petri net type definition. For the types deployed together with the ePNK, this information can be found in Sect. 3.5.

Note that labels, in principle[5], can have line-breaks. Since pressing the ENTER button will finish the editing of a label, however, a line-break is inserted to a label by pressing CNTRL-ENTER while editing the label.

3.4.3 Sub-pages

The graphical editor allows you also to create other pages on that page, which we call *sub-pages* in order to avoid confusion. This can be done with the "Page" tool, in the very same way places or transitions are created. In the graphical editor, sub-pages are graphically represented as rounded rectangles (or squares).

[5]That is, if the legal syntax of a specific Petri net type does allow it

It is also possible to open a graphical editor on a sub-page with via a pop-up menu on the right mouse button: "ePNK" → "Start GMF Editor on Page" (as we have seen it for the tree-editor). Therefore, the tree-editor could be used for creating the top-level pages of the net only; all the sub-pages could be created by the graphical editor. But navigation is much easier in the tree-editor; this is why you would probably want to use the tree- editor for navigating and opening sub-pages further down in the tree-hierarchy. It is recommended not to create sub-pages in the tree editor, since they would not have a position in the graphical editor. Still, it is possible and the graphical editor would show these pages (as well as other objects created in the tree-editor) in the top-left corner, when it is opened with the graphical editor for the first time. Then, you could move it to a better position.

What is more important about pages is how to deal with their labels. All the type-specific labels will be shown as page labels within the page. The name, however, will be shown as a label attached to that page on the super-page.

> **Note:** There might be some Petri net types that have some labels for pages that, like the name, should rather be shown as a label attached to the page on the super-page than as a page label within that page. The mechanism for defining Petri net types could provide a way of identifying which is which. But, right now, we do not have any example for such labels. Therefore, all label of a page except for the name are considered to be page labels in any Petri net type. Once we learn about a reasonable example, it would be easy to extend the type definition mechanism in such a way. Please, contact us, if you have a good example.

3.5 Petri net types

In this section, we give an overview of the Petri net types that are deployed together with the ePNK. Currently, these are P/T-Systems (PTNet) and high-level Petri nets (HLPNG). And there is the empty type (Empty), which, however, does not contain any concepts in addition to the PNML core mode; therefore, we do not need to discuss this here. The empty type was introduced to explicitly indicate, that there are no Petri net type specific extensions.

Actually, HLPNGs come in different levels or kinds: “dot nets”, which is a representation of P/T-Nets in high-level nets; basically, “dot nets” are restricted to the sort “DOT” and a minimal version of operators on them; “symmetric nets”, which is a restriction to some special finite sorts and a limited set of operations; and the full version of high-level nets. The kind of a HLPNG can be changed by selecting the HLPNG type in the tree editor and selecting the kind in the properties view (identified by the ISO/IEC 15909-2 URI). For a detailed discussion of the legal constructs of the different kinds of HLPNG, we refer to [5]. Note that, in contrast to the Petri net type, the kind of a HLPNG can be changed anytime, since the kind of HLPNG concerns the validation only. The PNML syntax is the same.

3.5.1 PTNet

We start with explaining PTNets. In Sect. 2.2.2 we have already seen the additional features of PTNets, which are the initial marking for places and the inscription for arcs. Both labels are simple labels, which means that it will be checked right after editing whether the label is syntactically correct (see Sect. 3.4.2); if it is not correct the the value will be reverted to the value it had before.

The marking of a place must be a non-negative integer in any reasonable representation[6]. The arc-inscription is similar, just that it must represent a positive integer (i. e. must be greater than 0).

Moreover, PTNets have the restriction that arcs may only run from a place to a transition or from a transition to a place, which will be enforced in the graphical editor[7]. Actually, the constraint is slightly more complicated due to reference nodes: We can connect place-like nodes (PlaceNodes) with transition-like nodes (TransitionNodes) and vice versa; but semantically, i. e. when flattening reference nodes, this amounts to the above condition.

3.5.2 HLPNG

HLPNGs are much more involved than P/T-Nets and we cannot explain them in all details here. For a detailed motivation and full account on what HLPNGs are, we refer to ISO/IEC 15909-2 or [5].

For HLPNGs, there are the following labels (in addition to names):

[6]For those who want to bother with the technical details, it is any String that would be accepted by the Java `Integer.parseInt()` method as a number and evaluates to a number greater or equal than 0.

[7]In the tree editor, illegal arcs could be created, but this would not pass validation.

Declaration A *declaration* is page label, which are used to define *variables*, *sorts*, and *operators*, which can then be used in the other labels. Every page can have any number of declarations and, within a single declaration, different kinds of declarations can be mixed. Note that all declarations are global (known in the complete Petri net), even though it is attached to a specific page.

Type A *type* is a label that is associated with a place. Every place must have exactly one type label which denotes the sort (which can be built from built-in sorts and sort constructs or from user defined sorts) of the tokens on that place.

HLMarking A *marking* is a label that is attached to a place and indicates the places initial marking. The marking is represented by a ground-term[8], which must be a multiset over the places type. Note that this label may be omitted, in which case the initial marking is considered to be empty. But there can be at most one label of this kind.

Condition A *condition* is a label that can be attached to a transition. The condition is a term of type boolean and can contain variables. There can be at most one condition, and if it is missing, it is assumed to be true.

HLAnnotation An *arc annotation* is also a term that may contain variables. The term must be a multiset term over the type of the place to which the arc is attached. Every arc should have exactly one arc annotation[9].

> **Note:** ISO/IEC 15909-2 also allows declarations directly attached to the net. This is not supported by the ePNK, because this breaks one of PNML's own principles and could not be shown graphically. If there was strong need for this feature, it could be implemented though. Please, let us know.

All labels of HLPNGs are structural labels (see Sect. 3.4.2), which means that the user can edit it and leave it syntactically incorrect. Of course, this will not pass validation; it is also possible to save nets with incorrect labels and load them again, so that the labels can be corrected another time. Since

[8] A ground-term is a term that does not contain variables.

[9] Actually, the ISO/IEC 15909-2 would allow that this label is missing. This does not make much sense though, since in most cases there is no reasonable standard interpretation if the label is missing.

labels can refer to symbols that are defined other places, it is also important to link these labels before validation (see Sect. 3.3.4), which can be done by the pop-up menu on the net in the tree-editor: "ePNK"→"Link Labels (globally)".

PNML does actually not define a concrete syntax for declarations and terms. This is up to the tool. Therefore, the ePNK comes with its own concrete syntax, which resembles the one of CPNTools, but is not identical! Before going into the details of the syntax, we briefly discuss some examples.

The following shows several declarations of variables, sorts and operators. Each of them could be in a separate declaration label, but they could also be contained in a single declaration:

```
vars
  x:NAT;

sorts
  A = MS(BOOL);

ops
  f(x:INT, y:INT) =  x * y,
  g() = 1;

sorts B = (A*INT), C = (B*B);
```

First, a variable x of built-in sort NAT is defined. Then a user-defined sort A is defined, which is a multiset over the built-in sort BOOL. Then, two named operations are defined, f and g. f takes two parameters of type INT and g does not have parameters. Note that named operations, basically, are abbreviations and, therefore, do not allow any recursion (see [5]). In the end, two other user-defined sorts are defined: B is a product of A and the built-in sort INT, and C is a pair over sort B. Note that also for sort declarations, recursion is not allowed.

The right-hand side of the sort declarations above give you an idea of the syntax for sorts already. There are some built-in sort like BOOL, INT, NAT, POS and DOT. From these, we can built products or multiset sorts.

Here are some examples of terms (using the above declarations):

```
 x‘f(x,x) ++ 1‘x ++  x‘g() ++  1‘5

 1‘(dot,1) ++ 1‘(dot,1*1)
```

```
x > 1 and x < 5
```

The first is a multiset term over the sort INT, which could be used in arc inscriptions (if the attached place is of type INT). The second is a ground term over the product of built-in sort DOT with INT, where DOT is a sort that represents a type with a single element dot. The last term is a term of sort BOOL, which could be a condition.

The precise syntax is defined by the following grammar (that actually is a simplified version of the grammar that was used for generating the parser). The terminals ID, INT, NAT, STRING in this grammar represent legal identifiers and legal representations of integer numbers, non-negative integer numbers and string constants.

Listing 3.1 shows the part of the grammar for declarations. Listing 3.2 shows the part of the grammar for terms. Note that this part of the grammar is ambiguous for making it a bit more readable. The ambiguities is resolved by assigning a binding priority to the different operators – moreover all operators are left-associative. Every line in the declaration of BinOp represents operators on the same level of priority, where the first line has the least binding-power and the last the highest. The unary operators (actually there is only one) have the highest binding power of all. Note that there are also some operators like the cardinality, which use circumfix notation: if m is some multiset `|m|` denotes the cardinality of that multiset. This operator has the same binding power as parentheses.

Listings 3.3 and 3.4 show the part of the grammar for built-in sorts and constants. Note that every number constant will implicitly be assigned the best fitting sort: INT, NAT, or POS. If a positive integer, say 5 should have the type INT instead, this can be represented by 5:INT, which is like a type cast in object-oriented programming languages.

In addition to these syntactical constraints, the terms must also be correctly typed, which we do not discuss here in detail.

For HLPNGs, there are many constraints. Like for PTNets, arcs may only run from places to transitions or from transitions to places. All of the other additional constraints concern the correctness of the labels of HLPNGs. The following list gives an overview:

1. Every place must have a (correct) type.

2. Every declaration must be syntactically correct and correctly typed.

3. All declarations must properly resolve (must not be recursive).

Listing 3.1: Grammar for declarations

```
Declarations :
    ( 'sorts' SortDecl ( ',' SortDecl )* ';'                    |
      'vars'  VariableDecl ( ',' VariableDecl )* ';'            |
      'ops'   OperatorDecl ( ',' OperatorDecl )* ';'            |
      'sortsymbols' ArbitrarySort ( ',' declaration )* ';'      |
      'opsymbols' ArbitraryOperator ( ',' ArbitraryOperator )*
    )*;

SortDecl :
    NamedSort;

NamedSort :
    ID '=' Sort;

VariableDecl :
    ID ':' Sort;

OperatorDecl :
    NamedOperator;

NamedOperator :
    ID '(' ( VariableDecl ( ',' VariableDecl )* )? ')' '=' Term;

Sort :
    BuiltInSort | MultiSetSort | ProductSort | UserSort;

MultiSetSort :
    'MS' '(' Sort ')';

ProductSort :
    '(' ( Sort ( '*' Sort )*)? ')';

UserSort :
    ID;

ArbitrarySort :
     ID;

ArbitraryOperator :
     ID ":" ( Sort ("," Sort )* )? "->" Sort;
```

Listing 3.2: Grammar for terms

```
Term :
    Term BinOp Term |
    UnOp Term       |
    BasicTerm;

BinOp :
    // all binary operators are left-associative
    'or' | 'implies'                      |  // lowest priority
    'and'                                 |
    '>' | '>=' | '<' | '<=' | 'contains' |  // all comparison ops
          '<r' | '<=r' | '>r' | '>=r' |     // on same level
          '<p' | '>p'  |                    //
          '<s' | '<=s' | '>s' | '>=s'    |  //
    '==' | '!='                           |
    '++' | '--'                           |
    '‘'                                   |
    '+' | '-'                             |
    '*' | '**' | '/' | '%'                ;  // highest priority

UnOp :
    'not' ;     // higher priority than all binary operators

BasicTerm :
    Variable                  |
    UserOperator              |
    OtherBuiltInOperator      |
    BuiltInConst              |
    '(' Term ')'              |  // a sub-term in parentheses
    '(' Term ( ',' Term )+ ')';  // a tuple

Variable :
    ID;

UserOperator :
    ID '(' ( Term (',' Term )* )?  ')' ;

OtherBuiltInOperator  :
    '|' BasicTerm '|' |  '#(' Term ',' Term  ')'  |
    CyclicEnumsBuiltInOperator  | PartitionsBuiltInOperator   |
    StringsBuiltInOperator | ListsBuiltInOperator;
```

Listing 3.3: Grammar for sorts and constants (1)

```
BuiltInSort  :
    Dot | Boolean | Number FiniteEnumeration | CyclicEnumeration |
    FiniteIntRange | StringSort | ListSort ;

BuiltInConst  :
    DotConstant | BooleanConstant | MultisetConstant | NumberConstant |
    FiniteIntRangeConstant | StringConstant | ListConstant ;

MultisetConstant  :
    'all' ':' Sort  |
    'empty' ':' Sort;

Dot :
    'DOT';

DotConstant :
    'dot';

Boolean :
    'BOOL';

BooleanConstant :
    'true' | 'false';

Number :
    'INT' | 'NAT' | 'POS' ;

NumberConstant :
    INT (':' Number)?;
```

Listing 3.4: Grammar for sorts and constants (2)

```
FiniteEnumeration : 'enum' '{' ID ( ',' ID)* '}' ;

CyclicEnumeration : 'cyclic' '{' ID (',' ID)* '}' ;

CyclicEnumsBuiltInOperator :
     'succ' '(' Term ')'   |   'pred' '(' Term ')' ;

FiniteIntRange : '[' INT '..' INT ']' ;

FiniteIntRangeConstant : INT FiniteIntRange ;

Partition :
     'partition' Sort 'in' ID
       '{' PartitionElement ( ';' PartitionElement )* '}';

PartitionElement : ID ':' Term  ( ',' Term )* ;

PartitionsBuiltInOperator : 'partition' ':' ID '(' Term ')';

StringSort : "STRING" ;

StringsBuiltInOperator :
     "concatstring" "(" Term "," Term ")"      |
     // note that we do not have append (does not make sense)
     "stringlength" "("  Term  ")"             |
     "substring" ":" NAT "," NAT "(" Term  ")" ;

StringConstant : STRING ;

ListSort :  "LIST" ":" Sort;

ListsBuiltInOperator :
     "concatlists" "(" Term "," Term ")"                |
     "appendtolist" "(" Term "," Term ")"               |
     "listlength" "("  Term  ")"                        |
     "sublist" ":" NAT "," NAT "(" Term  ")"            |
     "memberat" ":" NAT "(" Term  ")"                   |
     "makelist" ":" Sort "(" (Term ( "," Term)* )? ")"  ;

ListConstant : "emptylist" ":" Sort ;
```

4. Every term (in markings, arc annotations, and conditions) must be syntactically correct and correctly typed.

5. The marking of a place must be a ground term and must be a multiset over the sort of the place.

6. The arc annotation must be a term that is a multiset over the place's sort.

7. Every condition must be a term of sort BOOL.

8. All declarations must have a distinct name (actually, this causes a warning only since this is a condition on concrete syntax, which is not part of PNML).

9. Parameters of an operation declaration should have distinct names (actually, this causes a warning only since this is a condition on concrete syntax, which is not part of PNML).

3.6 Limitations and pitfalls

The current version of the ePNK (0.8.1) has some limitations and some problems, which will be discussed in this section in order to avoid some unpleasant surprises. If you identify other issues or problems, please, let us know.

3.6.1 Dirty-flag and saving

Technically, the tree-editor and the graphical editor for pages are different editors. Up to now, they are working together, but they are not tightly integrated yet. This has the effect that the "dirty-bit' of the editor (indicated by a star in the editor tab) is not always up to date and is reliable. It is recommended that all save operations are done from the tree-editor and that the tree-editor is always open and the last to be closed. Before finally closing the tree-editor, make some minor change (e. g. in a net's name) and then save it. Do not close the tree-editor on a net when there are still graphical editors open[10].

In particular when a PNX-file is saved and closed with a graphical editor open, you will not be able to load it again. If you experience that problem,

[10] I will, eventually, implement a control that prevents the tree-editor from being closed as long as there are graphical editors and also make sure that the dirty-bit is properly updated. But, this does not have the highest priority on my worklist right now.

you can work around this by opening the corrupted file in a text-editor and delete all the diagram information of a graphical editor[11].

3.6.2 Graphical features

The graphical editor supports many features, such as different fonts, and colours for labels, colours and linewidth for nodes, and different versions of curved arcs. Up to now, this information is not stored in the PNML files (and some of it is not even supported by PNML at all).

Right now, the only graphical information that is stored in the saved PNML or PNX file is the following:

- Position and size of nodes
- Position (resp. relative position) of labels (the size of labels is not supported by PNML).
- Intermediate points of arcs (but only as polyline; the different kinds of curved lines of GMF are not supported by PNML anyway; the bezier curves that are supported by PNML are not supported by GMF).

Therefore, you should not put too much effort in unsupported features, since they will be gone when you open the page next time. It is planned, to implement a toolspecific feature that would keep the graphics exactly as it was edited in the ePNK, though. But this would not be usable by other tools, since it is tool specific.

The ePNK will read all graphical attributes that are supported by PNML; but, except for the ones discussed above, they are not yet shown graphically. And some of the graphically attributes, like CSS attributes for fonts, colours, etc. are not checked for validity; they will be written exactly the same way they were, when loading the PNML file (see also Sect. 3.6.5).

3.6.3 Petri net types

As mentioned in Sect. 3.3.2, the type of a Petri net should be added right after the creation of a Petri net, and the type of the Petri net should never be changed later – except if you know exactly what you are doing. Otherwise, it could happen that produced PNML is invalid.

[11]I plan to implement a toolspecific extension for the ePNK, which can save the diagram information of GMF in the PNML model; this will fix this problem.

For HLPNGs, it is now problem to change its kind any time, since this kind has an effect on the validation only, but no effect on the serialisation of the net to a PNML file.

3.6.4 Wrapping labels

All labels in ePNK can have line-breaks. In the graphical editor, the line-breaks can be added by pressing the CNTRL and ENTER at the same time.

3.6.5 Graceful PNML interpretation

The PNML files that are produced by ePNK are PNML conformant as defined in ISO/IEC 15909-2. The only exception is, when some illegal graphical attributes are read from an existing PNML file; these attributes will not be touched by the ePNK, and therefore written again. But, if a PNML file was created by using the ePNK only and if it validated correctly, the saved file is PNML conformant.

The ePNK, however, is not a PNML validator. It reads PNML (and PNML-like) documents and writes them again in a graceful manner. This way, it is possible to save documents that do not properly validate and load them again. For example, when some element do not have ids, the references to these elements are stored as XPath references, which is not conformant to PNML. If the ids are added later, and validation is successful, this will produce conformant PNML documents again.

3.6.6 Deviation from PNML

There is, however, one feature of HLPNGs that the ePNK is not able to read. These are labels that are directly attached to the net and not to one of its pages, which we call *net labels*. Since net labels are against PNML's own principles and they do not appear to be important anyway, net labels are not supported by the ePNK for now[12].

The only way to read a PNML document with net labels with the ePNK right now is to open it in some text editor and add a pair of `<page>` and `<\page>` tags around all net labels. Then, the ePNK would be able to open it (with the original net labels on a separate page – without id and page name, which should be added then).

[12] If there is some evidence that this feature is important, the ePNK could be extended to support it – with some reluctance though.

Chapter 4

Developers' guide

In this chapter, we discuss on how to extend the ePNK, by defining new functionality, by defining new Petri net types, or by defining new tool-specific extensions. For all these extensions, the ePNK provides *extension-points* so that the extensions can be made without changing the actual code of the ePNK[1]. Actually, the ePNK does not even provide own extension-points for adding functionality: The existing eclipse extension-points are good enough for that for now[2].

We start with Sect. 4.2, which shows how to add some functionality to the ePNK, which could be implementing a model-checker, or some other analysis or verification function, it could be a function that reads a net in PNML and produces some net in some other format, or a function that generates a net that is stored in the PNML format.

In Section 4.3, we discuss how to add a new Petri net type to the ePNK. Simple net types can almost completely be generated from a model, for more complex ones, such as high-level Petri nets, a mapping to XML must be defined.

At last, we discuss how to add tool-specific information to the ePNK in Sect. 4.4.

All of these extensions will be discussed by the help of some examples, which are deployed together with the ePNK. In these examples, we assume that the reader is familiar to the main principles and ideas of eclipse, its plug-in architecture, and eclipse plug-in development (see for example, the

[1]Technically, you would not even need to see the code of the ePNK, but looking at it might help understand the ideas and principles behind the ePNK.

[2]Eventually, there might be some ePNK-specific extension-point for adding functionality for getting a more uniform "user-impression" of the added functionality. Since this is mostly a GUI-issue, this does not have the highest priority right now.

"Platform Plug-in Developer Guide" which is part of eclipse: "Help" → "Help contents" or one of the many eclipse resources [13, 4]); Sect. 4.1 will give a brief overview though. We go a bit more into the details of EMF and explain some of the steps that need to be done in EMF more explicitly, but it is also recommended to read up on some details on EMF [2] before starting with own development projects.

4.1 Eclipse: a development platform for the ePNK

As briefly discussed in Sect. 3.1 already, eclipse is an Integrated Development Environment (IDE). Here, we briefly explain how to set up the eclipse environment so that you can work on your own extensions and do the tutorials of this chapter – assuming that you have installed eclipse and the ePNK as explained in Chapter 1 already.

4.1.1 Installing the ePNK source projects

As a developer you would probably want the source code for the projects, though you would probably not need to change it, and its recommended not to do so. You can download a zip-file "ePNK-0.9.0.zip" conatining all projects with the source code from the ePNK homepage: `http://www2.imm.dtu.dk/~eki/projects/ePNK/`

In your eclipse development workspace, select "File" → "Import..." and, in the dialog, select "Existing Projects into Workspace" from category "General". In the opended "Import" dialog select "Select archive file" and press "Browse" to select the "ePNK-0.9.0.zip" you obtained from the ePNK homepage. Then press "Finish".

In your workspace you have now many plug-in projects, which are the basis for developing new plug-ins and, in particular, extensions of the ePNK. Therefore, this workbench is also called the *development workbench*. The reason for introducing an additional attribute to the term *workbench* here is that we will have another workbench once we start our tool with the new extensions, which is another instance of eclipse again: This workbench is called the *runtime workbench* since this is when the ePNK with your new extensions is running (and it will very much look like the original ePNK as discussed in Chapter 3). This runtime workbench can be started from the development workbench by "Run" → "Run Configurations..." and then selecting "Eclipse Application" and pressing the "New" icon and then "Run". Later, it would be enough to press the "Run" button in the tool bar. For now, however, we do not start the runtime workbench since we need to implement some new functionality first.

4.1.2 Installing Ecore Tools

As mentioned already, a major part of defining new Petri net types is creating ecore models with the new concepts of the Petri net type. There is a extension of eclipse "Ecore Tools SDK", which helps with creating the models, and installing this extension, will also install all other tools necessary for generating code from the models.

To install , start the Install dialog[3] select the standard Galileo update site and select "Ecore Tools SDK (...)" and follow through the installation process.

After restarting eclipse, you can check whether the installation was successful: Open the project `org.pnml.tools.epnk` and, within that project, open the folder `model`. The files with extensions ".ecore", ".ecorediag", and ".genmodel" should have special icons now. If you open click on "PNML-CoreModel.ecorediag", you should see a ecore model in a class diagram like graphical notation (resembling the PNML core model from Fig. 2.1).

4.2 Adding functions

In this section, we discuss how to add new functionality to the ePNK. To this end, the API that is generated from the PNML core model and the Petri net type definitions are discussed – giving you an idea of the general principles, which can be found in [2]. Via this API, you can access and also modify nets in a uniform way. Moreover, this section shows how to open, create and write PNML files from another program.

To this end, this section will discuss how to plug in functionality into the ePNK (or to eclipse in general) in three different ways.

- Section 4.2.1 shows how to implement a new eclipse view (cf. 3.1), which gives an overview of a PNML file that is selected in one of the eclipse resource explorer views.

- Section 4.2.2 shows how to implement a *wizard* for creating a PNML file (actually, we changed a wizard that was automatically created by the eclipse "new plug-in project wizard"). This wizard will create a simple PNML file with a P/T-System that contains a mutual exclusion

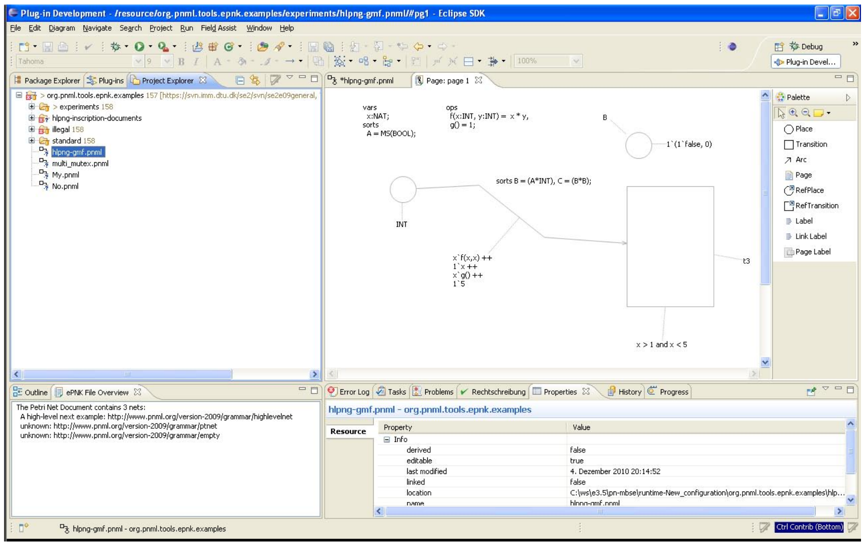
[3]Reminder: Select "Help" → "Install New Software...".

algorithm for a user selected number of agents. Each of these agents will be shown on a different page.

- Section 4.2.3 shows how to implement a simple pop-up menu on a selected Petri net (in the tree editor), which starts a model checker, asking the user for some formulas to be checked, and then checking the formulas on the net. Since model checking can take quite some time, the model checker will run in the background and can be aborted by the user. This uses eclipse's concept of jobs. On the side, this shows how to use some of eclipse's user dialog functions.

4.2.1 Accessing a PNML file and its contents: A file overview

In this section, we discuss how to implement a new (and very simple) view, that will give an overview of the contents of a file that is selected in the explorer. Figure 4.1 shows a screenshot of the result. For the selected file "hlpng-gmf.pnml" in the "Project Explorer", the "ePNK File Overview" in the bottom left, shows that the selected file is a Petri net document, which contains 3 Petri nets, a high-level net, a P/T-net, and an empty net – where the overview shows the respective "type tag" of PNML. The name of the first net is "A high-level next example"; the other two nets do not have a name.



```
The Petri Net Document contains 3 nets:
  A high-level next example: http://www.pnml.org/version-2009/grammar/highlevelnet
  unknown: http://www.pnml.org/version-2009/grammar/ptnet
  unknown: http://www.pnml.org/version-2009/grammar/empty
```

hlpng-gmf.pnml - org.pnml.tools.epnk.examples

Property	Value
Info	
derived	false
editable	true
last modified	4. Dezember 2010 20:14:52
linked	false
location	C:\ws\e3.5\pn-mbse\runtime-New_configuration\org.pnml.tools.epnk.examples\hlp...
name	hlpng-gmf.pnml

Figure 4.1: The ePNK with the "File Overview" view

This view and its functionality is implemented in the plug-in project `org.pnml.tools.epnk.functions.tutorial` and we will go through this project now[4]. The implementation of the view is contained in a single class: `PNMLFileOverviewView` in the package

`org.pnml.tools.epnk.functions.tutorial.overviewview`.

We briefly explain the general structure of this view class, which is is extracted in Listing 4.1, where we deleted imports and comment, which can be looked up in the source code. The class extends the eclipse `ViewPart`, which actually makes it a view and it implements the `ISelectionListener`, which allows our view to obtain the information on the current selection of the user in the workbench. Note that there is no explicit constructor. The reason is that the view will be set up via the `createPartControl` method: In the first three lines of that method, a viewer (which represents the content of that view) is initialized, and so-called providers will enable the view to properly show the contents. But, we do not do not discuss the details here. In the last two lines of the `createPartControl` method, our viewer registers itself with the eclipse selection mechanism as a *selection listener* and then creates the information that should be shown for the current user selection. We will discuss the respective method `selectionChanged` below. Note that there are two other methods. The `setFocus` just forwards the focus properly to the content of the view, once the view is focused. More important is the `dispose` method: the implementation of this method makes sure that our view removes itself as a selection listener once it is disposed (which typically would happen if the user decides to close the view).

Once the view has registered itself as a selection listener with the eclipse workbench, its `selectionChanged` method will be called whenever there is a change in the user's selection. In the implementation, of this method, the kind of the current selection is analysed and it is checked whether the first selected element is a file (i. e. whether it implements the interface `IFile`). If so, the method `getOverviewInfo` for computing the actual content of the file overview is called; this produces an array of Strings, which then will be set as the new content of that view – and, this way, shown to the user.

This `getOverviewInfo` is probably the most interesting part here, since it shows how to open and access a PNML or a PNX file (we do not even need to make a difference). The implementation of this method is shown in Listing 4.2 Up to line 7, it is checked whether the file extension is either "pnml" or "pnx" (the two file extensions, the ePNK has registered for the ePNK) and the path to that file is extracted and a URI is created. Actually,

[4]Note that this project also contains the implementation of the wizard in Sect. 4.2.2.

Listing 4.1: Class `PNMLFileOverviewView`: Infrastructure

```
package org.pnml.tools.epnk.functions.tutorials.overviewview;

import ...

public class PNMLFileOverviewView extends ViewPart
  implements ISelectionListener {

  private TableViewer viewer;

  public void createPartControl(Composite parent) {
        viewer = new TableViewer(parent);
        viewer.setContentProvider(new ArrayContentProvider());
    viewer.setLabelProvider(new LabelProvider());
    getSite().getPage().addSelectionListener(this);
    selectionChanged(null, getSite().getPage().getSelection());
  }

  public void setFocus() {
    viewer.getControl().setFocus();
  }

    public void dispose() {
        super.dispose();
        getSite().getPage().removeSelectionListener(this);
  }

  public void selectionChanged(IWorkbenchPart part,
      ISelection selection) {
    if (selection instanceof IStructuredSelection) {
      IStructuredSelection structured =
        (IStructuredSelection) selection;
      Object first = structured.getFirstElement();
      if (first instanceof IFile) {
        viewer.setInput(getOverviewInfo((IFile) first));
  }   }   }

  public String[] getOverviewInfo(IFile file) {
    ...
  }
}
```

Listing 4.2: Class PNMLFileOverviewView: Accessing the file

```
public String[] getOverviewInfo(IFile file) {
  String[] result = {"No ePNK file selected"};
  String extension = file.getFileExtension();
  if (extension != null &&
      (extension.equals("pnml" ) || extension.equals("pnx"))) {
    String path = file.getLocationURI().toString();
    URI uri = URI.createURI(path);

    ResourceSet resourceSet = new ResourceSetImpl();
    Resource resource = null;
    try {
      resource = resourceSet.getResource(uri, true);
    } catch (Exception e) {
      result[0] = "File could not be read.";
      return result;
    }

    List<EObject> contents = resource.getContents();
    if (contents != null && contents.size() > 0) {
      EObject object = contents.get(0);
      if (object instanceof PetriNetDoc) {
        PetriNetDoc document = (PetriNetDoc) object;
        List<PetriNet> nets = document.getNet();
        int no = nets.size();
        result = new String[no + 1];
        result[0] = "The Petri Net Document contains "
          + no + (no == 1 ? " net" : " nets:");
        no = 1;
        for (PetriNet net : nets) {
          String name = net.getName() != null ?
              net.getName().getText() : "unknown";
          String type = net.getType() != null ?
              net.getType().toString() : "unknown";
          result[no++] = "  " + name + ": " + type;
        }
      } else
        result[0] = "The file does not contain a PetriNetDoc.";
    } else
      result[0] = "The file does not contain any element.";
  }
  return result;
}
```

eclipse provides also user dialogs and file dialogs that would allow us to ask the user for a file name that would be returned as a URI; here we ab-used the selection mechanism and the file to get hold of some legal URI of a pnml or pnx file. Therefore, the code that comes now, could be used at any other point when a program wants to read and access some file, once we have a String with the path of the file: This starts with creating a *resource set* and, within that resource set creating a *resource* with the given URI, which is the first parameter of the `getResource` method; the second parameter indicates whether cross-references to other resources should be resolved lazily or not (which is not relevant here). Note that in EMF, a resource or file should always be accessed (and created, see Sect. 4.2.2 for more information) in this way via a resource set. After we successfully got the resource, we can obtain its content by the `getContents` method which returns a list of its top-level objects – in case of PNML, this this list should have length 1. Being defensive, we check whether the contents exists and whether its first element is an instance of PetriNetDoc. If so, we go systematically through all the contained nets, get their names and their PNML types and add a String with that information to the String array with the result. In the other cases, we return some error messages. Note that we do not need to close the file, or do anything else after we have obtained the information we need.

Let us have a closer look at how the contents of the Petri net document is accessed, once we have obtained a `PetriNetDoc` object. For any reference and attribute of the PNMLCoreModel there are respective getter and setter methods. For example, if we have a name label, the `getName` method will return the String with that name, and with `setName` we could set it – but we do not do that here. For attributes and features with a multiplicity greater than one, this is slightly different. For example a PetriNetDocument can contain many nets; therefore, `getNet` will return a list of nets, which then is iterated over to get the individual nets. And by adding a net to this list, this Petri net will be added to the Petri net document (see 4.2.2 for examples).

As stated above, the class `PNMLFileOverviewView` implements the "ePNK File Overview" as we have seen it in Fig. 4.1. But, if we just implement this class, it would not show up in eclipse, because eclipse would not know that it exists. In order to make the view known to eclipse we need to define it as an extension: This is done in the project's "plugin.xml". Double clicking on the plugin.xml file, will give you a convenient editor for defining and editing the extensions you want to define. Explaining the actual extensions are a bit easier with the XML fragment this editor produced. The fragment relevant for our overview view is shown in Listing 4.3. It says that an extension for

Listing 4.3: Defining the extension in "plugin.xml"

```
<extension
      point="org.eclipse.ui.views">
   <category
         name="ePNK"
         id="org.pnml.tools.epnk.views.category">
   </category>
   <view
         allowMultiple="false"
         category="org.pnml.tools.epnk.views.category"
         class="org.pnml. ... .overviewview.PNMLFileOverviewView"
         icon="icons/PetriNetDoc.gif"
         id="org.pnml.tools.epnk.extensions.tutorials.pnmloverview"
         name="ePNK File Overview">
   </view>
</extension>
```

extension point `org.eclipse.ui.views` is defined, which is a new eclipse view. The category defines, where and under which category the new view can be found. We define a category specific for the ePNK and then define the actual view to use it. The attributes of the views say, that a view of this kind can at most be open once, that it uses the above category, refers to the class which actually implements it, `PNMLFileOverviewView`, and defines an icon (used in the tab of that view) and a name for that view.

Note that in order to access some of the classes like `Resource`, `ResourceSet`, and some of the ePNK classes like `PetriNetDoc`, `PetriNet`, etc. in the implementation of the view, we would also need to define by which plug-in projects they are provided: If you open the file plugin.xml, you will find these projects in the tab "Dependencies". But this is a more technical issue, which we do not go into the details.

Now, you could start the runtime-workbench and then the "ePNK File Overview" view could be opened by the user by "Window"→ "Show View"→ "Other..." and then selecting "ePNK File Overview" in the category "ePNK". This will show the view in the workspace as shown in Fig. 4.1.

4.2.2 Writing PNML files: Generating a multi-agent mutex net

Next, we will discuss how to create new files and fill them with some content. In typical applications, the contents might come from a file in a format of another specific tool, which should be converted to PNML. In our example, however, we programmatically generate a Petri net: my favourite semaphore mutex example. To make it slightly more interesting, the number of agents competing for the semaphore is a parameter. This function is implemented as an eclipse wizard and it was implemented by creating a new file wizard for pnml files automatically by the eclipse "New Plug Project" wizard choosing the "custom plug-in wizard" with the choice of the "New File Wizard" in the "Template Selection" dialog. But, this does not need to bother you too much. If you are interested in the manual changes made to the automatically generated code, you will find all the manual changes in the two classes in the package `org.pnml.tools.epnk.functions.tutorials.wizards` enclosed by comments like `// eki: ...` .

In the rest of this section, we focus on the explanation of the parts of the implementation that are concerned with the file creation. This functionality is implemented in the method `createPNMLFile(String path, int number)` of the class `MultiAgentMutexNetWizard` in package

`org.pnml.tools.epnk.functions.tutorials.wizards`

in the plug-in project `org.pnml.tools.epnk.functions.tutorials`. The parameter `path` is a String representation of a path to the file that should be created. The parameter `number` is the number of agents that should be created in the mutex net that will be generated.

Creating a Petri net programmatically is quite simple, but code intensive. Therefore, we have split up the creating process into several parts for the different elements, which will be discussed top-down from creating the document, the net, its pages, and the places, transitions, reference places, and arcs on them. We discuss these methods one after the other – and omitting some boring ones in the end (you will find all details in the source code). Listing 4.4 shows the method that creates the file. First, it calls the method `createPetriNetDoc` that creates the Petri net document, which is discussed later. This is the content of the file that we want to write. Then, we convert the path into an URI. Then, we use the resource set again – this time to actually create the file. Surprisingly, enough this is already all we need to do. At this point, we can add the content to the resource. Note that it does not even matter whether the resource is a PNML file or a PNX file – eclipse will, dependent on the file extension, chose the right implementation

Listing 4.4: Method `createPNMLFile(String path, int number)`'

```
public void createPNMLFile(String path, int number) {
  PetriNetDoc doc = createPetriNetDoc(number);

  final URI uri = URI.createURI(path);
  ResourceSet resourceSet = new ResourceSetImpl();
  final Resource resource = resourceSet.createResource(uri);
  EList<EObject> contents = resource.getContents();
  contents.add(doc);
  try {
    resource.save(null);
  } catch (IOException e) {
    // Do nothing for now if file could not be saved.
  }
}
```

of the resource so that either a PNML file or a PNX file is written once we save the resource in the end. But, the wizard was configured in such a way that the user can only chose the "pnml" extension.

Adding the contents follows the same principle we have discussed already. With `getContents` we get a list of EMF objects (which would be empty); then we add the Petri net document to this list. The only thing left is to save the resource, which is done by calling the `save` method. Note that the save method has a parameter, that could be used to configure the way a file is saved. But, `null` is fine here – and you should only change this, if you know exactly what you are doing.

So, let us dive a bit deeper: into the method `createPetriNetDoc`, which takes one parameter only – the number of agents. This method is shown in Listing 4.5. In the second line, a new Petri net document is created. Note that this is not done with the usual `new` construct. Rather, the factory for the PNML core model which is obtained by `PnmlcoremodelFactory.eINSTANCE` is used for this purpose. It is part of the EMF philosophy that we actually should not know anything about the actual implementation of classes. And EMF strongly recommends to create new objects only via these factories. Note that the new net and its type are also created by factories – since the type is plugged in, a the factory of that plug-in is used for this purpose `PtnetFactory` resp. its instance. After creating the net, its id is set, by the `setId` method. Note that this could be any string, but it is our responsibility to make sure that all id's are different (if we create them programmatically).

Listing 4.5: Method `createPetriNetDoc(int number)`’

```java
public PetriNetDoc createPetriNetDoc(int number) {
  PetriNetDoc doc = PnmlcoremodelFactory.
    eINSTANCE.createPetriNetDoc();

  PetriNet net = PnmlcoremodelFactory.eINSTANCE.createPetriNet();
  net.setId("n1");
  doc.getNet().add(net);
  PetriNetType type = PtnetFactory.eINSTANCE.createPTNet();
  net.setType(type);

  Name nameLabel = PnmlcoremodelFactory.eINSTANCE.createName();
  nameLabel.setText("Mutual exclusion");
  net.setName(nameLabel);

  Page page = createPage(type, "pg0", "semaphor page");
  EList<Page> pages = net.getPage();
  pages.add(page);

  Place semaphor = this.createPlace(
      type, "semaphor", "semaphor", 1, 380, 140);
  page.getObject().add(semaphor);

  for (int i=1; i<= number; i++) {
    page = createAgentPage(type, semaphor, i);
    pages.add(page);
  }

  return doc;
}
```

Then the net is added to the list of nets of that document: to this end, we get the list of all nets of the document via `getNet` on which we call the `add` method. There is no way to directly add a net to a document. The type of the net can, again, be set with the `setType` method, since the type does not allow for multiple values.

After that, a name label is created, its text value is set, and the name label is added to the net.

Next a new page is created by calling a separate method, which is added to the list of pages of the net, and the place semaphore is created as the single object on this place. To this end, we use the `createPlace` method.

In the for-loop at the end of the method for each agent, there will be created one page with the net for the each agent.

All the other methods follow the same principles, and there is not too much interesting to see in them. Therefore, we finish with discussing the method `createAgentPage`, which creates 3 places, one reference place (referring to the semaphore that was created on the first page above), 3 transitions, and 8 arcs. What makes this method a bit more interesting is the graphical information that is added to the arcs: some intermediate point, which makes the net look a bit nicer. If you have a closer look at the `createTransition`, `createPlace`, and `createRefPlace` you will find similar constructs for defining the position and size of the nodes, and the position of the labels associated with them. But this should be straight forward, since this follows the exact principles of ISO/IEC 15909-2 (see [6]).

In the runtime workbench (or a version of the ePNK in which this plug-in is installed), you could invoke this function as follows: Go to the resource explorer – or any other explorer – of the workbench, press the right mouse button and select "New"→"Other...", select "Multi-agent Mutex Net Wizard" in the category "ePNK". Then, a dialog opens in which you can chose a folder[5] ("container") in which this file should be created, a "file name" (which must have extension "pnml"), and the number of agents. Note that, normally, the file creation wizard would overwrite existing files. This "multi-agent" wizard, however, was changed in such a way that existing files won't be overwritten accidentally.

4.2.3 Long-running functions: A model checker

In this section, we discuss the implementation of a model checker for P/T-Nets, which are interpreted as Condition/Event-Systems here. To this end

[5]If you have selected exactly one folder when you invoke the wizard, the fields of this dialog will be pre-set.

Listing 4.6: Method `createAgentPage()`'

```
public Page createAgentPage(PetriNetType type, Place sem, int i) {
  Page page = createPage(type, "pg"+i, "agent"+i);

  Place idle = createPlace(type, "idl"+i, "idl"+i, 1, 100, 220);
  Place pending = createPlace(type, "pen"+i, "pen"+i, 0, 100, 60);
  Place critical = createPlace(type, "cri"+i, "cri"+i, 0, 300, 140);
  RefPlace semRef = createRefPlace("sem"+i, "sem", sem, 380, 140);

  Transition t1 = createTransition(type, "t1."+i, "t1."+i, 40, 140);
  Transition t2 = createTransition(type, "t2."+i, "t2."+i, 220, 60);
  Transition t3 = createTransition(type, "t3."+i, "t3."+i, 220,220);

  Arc a1 = createArc(type, "a1."+i, idle, t1);
  Arc a2 = createArc(type, "a2."+i, t1, pending);
  ...
  Arc a6 = createArc(type, "a6."+i, t3, idle);

  Arc a7 = createArc(type, "a7."+i, semRef, t2);
  Coordinate coordinate =
    PnmlcoremodelFactory.eINSTANCE.createCoordinate();
  coordinate.setX(300);
  coordinate.setY(60);
  ArcGraphics arcGraphics =
    PnmlcoremodelFactory.eINSTANCE.createArcGraphics();
  arcGraphics.getPosition().add(coordinate);
  a7.setGraphics(arcGraphics);

  Arc a8 = createArc(type, "a8."+i, t3, semRef);
  ...
  a8.setGraphics(arcGraphics);

  EList<Object> contents = page.getObject();
  contents.add(idle);
  contents.add(pending);
  ...
  contents.add(t3);
  contents.add(a1);
  ...
  contents.add(a8);

  return page;
}
```

we use a simple library for symbolic model checking that was developed for teaching purposes: *Model Checking in Education* (*MCiE*)[6]. This library is deployed as part of the ePNK tutorials.

Within this developers' guide, we will not go into the details of model checking and its theoretical foundation, since this is not the point of this tutorial at all. For more information on model checking, we refer to a standard text book on model checking [3]. The point of this tutorial is to show how some function can be installed as a *pop-up* menu and *action* on a Petri net that is open in the tree editor. The actual action can be a function that might take some time and, therefore, should not block the graphical user interface of eclipse. To this end, the ePNK provides a way that makes it easy to use the eclipse *jobs*, which are running in the background – but of course provide the possibility to show a result.

The model checking functionality is implemented in the plug-in project `org.pnml.epnk.functions.modelchecking`. The main function (the actual model checking job) is implemented in the class `ModelcheckingJob` in package `org.pnml.epnk.functions.modelchecking.action`. The action initiating the model checking job is `ModelcheckingAction` in the same package.

Since the class `ModelcheckingAction` and the way it is integrated to the ePNK is quite simple, we start with explaining that first. It is shown in Listing 4.7. This class extends the `AbstractEPNKAction`, which is an ePNK convenience class implemented to make it easy to add a new action. It overwrites two methods, `isEnabled` and `createJob`. The method `isEnabled` checks whether the action is applicable for a Petri net. In our example, it is checked whether the Petri net has a type set and whether this type is `PTNet`. The other method `createJob` creates the actual job, which is an instance of `ModelcheckingJob`, with a Petri net and a defaultInput (for the user dialog). This class extends the ePNK's convenience class `AbstractEPNKJob` and will be discussed later.

In order to make the action `ModelcheckingAction` know to eclipse and appear in the popup menu in the "ePNK" category, we need to define an extension. Listing 4.8 shows the part of the "plugin.xml" file that defines this extension.

At last, we have a look at the class `ModelcheckingJob`, which is implementing the user dialogs (asking the user for temporal formulas), converting the Petri net into ROBDDs, doing the actual model checking, and showing the result to the user again. In addition to the constructor, we need to im-

[6]see `http://www2.cs.uni-paderborn.de/cs/kindler/Lehre/MCiE/`

Listing 4.7: The action class `ModelcheckingAction`'

```
package org.pnml.tools.epnk.functions.modelchecking.action;

import
    org.pnml.tools.epnk.actions.framework.jobs.AbstractEPNKAction;
import org.pnml.tools.epnk.actions.framework.jobs.AbstractEPNKJob;

import org.pnml.tools.epnk.pnmlcoremodel.PetriNet;
import org.pnml.tools.epnk.pnmlcoremodel.PetriNetType;
import org.pnml.tools.epnk.pntypes.ptnet.PTNet;

public class ModelcheckingAction extends AbstractEPNKAction {

  @Override
  public boolean isEnabled(PetriNet petrinet) {
    if (petrinet != null) {
      PetriNetType type = petrinet.getType();
      return type != null && type instanceof PTNet;
    }
    return false;
  }

  @Override
  protected AbstractEPNKJob createJob(PetriNet petrinet,
      String defaultInput) {
    return new ModelcheckingJob(petrinet,defaultInput);
  }

}
```

Listing 4.8: Defining the popup action for the model checking action

```
<extension
    point="org.eclipse.ui.popupMenus">
  <objectContribution
      id="org.pnml.tools.epnk.functions.modelchecking.contribution1"
      objectClass="org.pnml.tools.epnk.pnmlcoremodel.PetriNet">
    <menu
        id="org.pnml.tools.epnk.actions.standardmenu"
        label="ePNK"
        path="additions">
      <separator
          name="group1">
      </separator>
    </menu>
    <action
        class="org.pnml.tools.epnk. ... .action.ModelcheckingAction"
        enablesFor="1"
        id="org.pnml.tools.epnk.functions.modelchecking"
        label="Model checker"
        menubarPath="org.pnml.tools. ... .standardmenu/group1">
    </action>
  </objectContribution>
</extension>
```

plement (overwrite methods of `AbstractEPNKJob`) the following methods: `prepare()`, `getInput()`, `run()`, `showResult()`, and `canceling()`. Here is what these methods do:

Constructor: Sets up all the data structures needed during the job; typically, this will be storing the default input. In our model checker example, we also set up some mappings, for mapping places of the Petri net to variables of the MCiE library, and mappings from transitions to formulas defining their behaviour, and some other information. The code of the constructor is shown in Listing 4.9.

Listing 4.9: The constructor of `ModelcheckingJob`'

```
public ModelcheckingJob(PetriNet petrinet, String defaultInput) {
  super(petrinet, "ePNK: Model checking job");
  if (defaultInput != null) {
    defaultformula = defaultInput;
  }

  place2variable = new HashMap<Place,Variable>();
  place2primedvariable = new HashMap<Place,Variable>();
  transitions    = new Vector<Formula>();
  placeNames    = new HashSet<String>();
  duplicateNames = false;
}
```

`prepare()`: This method is handling the user dialogs before the actual job starts. In our case, it asks the user for some CTL-formulas and, also allows the user to correct the input, if the formulas are syntactically incorrect – or to abort the action. The code for this user dialog is shown in Listing 4.10. Since this is the standard way of doing this in eclipse, we do not go into the details of this part though. The only relevant part for the ePNK is that the job will not be continued, if the `prepare()` method returns false – in the implementation of the model checking job, this is done, when the user presses cancel in one of the dialogs (line 24/25 and line 33/34).

In our model checker job, the `prepare` method will try to convert the Petri net into formulas defining the behaviour of the transitions and the initial marking. And on the way, it will be checked whether there are duplicate names of places, so that a warning can be issued. Listing 4.11 shows the part of the `prepare` method converting the initial

Listing 4.10: The user dialog of the `prepare` method

```
...

InputDialog dlg = new InputDialog(
    null,
    "ePNK: Model checker",
    "Enter a comma separated list of temporal formulas please:",
    defaultformula,
    null);
dlg.open();

if(dlg.getReturnCode()!=Window.OK)
  return false;

defaultformula = dlg.getValue();

do {
  try {
    Parser parser = new Parser(new StringReader(defaultformula));
    formulas = parser.parseFormulaList();
    parser.parseEnd();
  } catch (Exception e) {
    formulas = null;
    dlg = new InputDialog(
        null,
        "ePNK: Model checker",
        "Syntax error in formula: \n\r" +
        e.toString() + "\n\r" +
        "Fix the error please or press cancel:",
        defaultformula,
        null);
    dlg.open();

    if(dlg.getReturnCode()!=Window.OK) // Didn't click on OK!
      return false;
    defaultformula = dlg.getValue();
  }
} while (formulas == null);
```

marking into a state formula. The basic idea is that in this formula

Listing 4.11: Building the formula for the initial marking (in `prepare()`)

```
FlatAccess flat = new FlatAccess(getPetriNet());

init = new Constant(1);
for (org.pnml.tools.epnk.pnmlcoremodel.Place p : flat.getPlaces()) {
  if (p instanceof Place) {
    Place place = (Place) p;
    registerPlace(place);

    PTMarking marking = place.getInitialMarking();
    if (marking != null && marking.getText().getValue() > 0) {
      init = new BinaryOp(BinaryOp.AND,
          init,
          place2variable.get(place));
    } else {
      init = new BinaryOp(BinaryOp.AND,
          init,
          new UnaryOp(UnaryOp.NOT,place2variable.get(place)));
    }
  }
}
```

a variable corresponding to the place occurs exactly once. It occurs negated, if the place is not marked and it occurs without negation, if the place is marked (with at lest one token[7]). All these negated variables will be connected by and-operations and the formulas are represented in MCiE's data structure. What is more interesting here is that the ePNK provides a way to access a net that consist of pages with reference nodes in a flattened way. This convenience class of the ePNK is called `FlatAccess`, which can be initialized with a Petri net of any type. Then the instance `flat` is used to get all places of the net, independently of the pages they occur on. Likewise, `flat` provides methods to access all the transitions and to get all the input and output arcs of a place or transition (including the ones of the reference nodes referring to them). This way, it is easy to obtain the pre- and post-sets, without being bothered with the page structure. For some more examples of the use of these methods, you can have a look into

[7]Remember that we abuse P/T-Nets for representing Condition/Event-Systems.

the code that converts transitions into formulas, which however is not discussed here.

The last part of the prepare method, is converting the formulas into OBDD-representation and creating a transition system out of these formulas. This is shown in Listing 4.12. Again, this is specific to MCiE. But, there are two parts that are important for the `prepare` method in general: With `this.setName()`, we can give the job a specific name, which is used in eclipse's jobs view. In our example, we say that it is a model checking job, add the name of the net and the formula which the user entered. The last important part is that the `prepare` method returns `true` in order to indicate that the preparation successfully terminated, and the actual job can be run (in the background) now.

Listing 4.12: Finishing the `prepare` method

```java
Name name = getPetriNet().getName();
String netref = "";
if ( name != null && name.getText()!= null) {
  netref = " on net " + name.getText();
}

this.setName("Model checking job" + netref +": " + defaultformula);

Context context = new Context();
ROBDD is = init.toROBDD(context);
ROBDD ts[] = new ROBDD[transitions.size()];
ChangeSet css[] = new ChangeSet[transitions.size()];

for (int i = 0; i< ts.length; i++) {
  ts[i] = transitions.get(i).toROBDD(context);
  css[i] = new ChangeSet(context);
  transitions.get(i).addChangedVariables(css[i]);
}
transitionsystem = new Transitionsystem(context,is,ts,css);

return true;
```

`getInput()` This method is called by the action, to get and store as default for the next call, the user input. In our case, the formula that was entered by the user during the prepare phase (in its String representation as entered by the user) is returned.

`run()` This method implements the part of the job that will be run in the background. In our case, this is the model checking job. This methods is actually quite simple (most of the programming work lies in the preparation). It is shown in Listing 4.13. But, it is the most

Listing 4.13: The `run` method

```
protected void run() {
  result = "Model checking results:\n\r";
  for (int i = 0; i < formulas.length; i++) {
    ROBDD obdd = formulas[i].toROBDD(transitionsystem);
    result = result + " " + formulas[i] + ": " +
      transitionsystem.isValid(obdd) + "\n\r";
  }
}
```

computation intensive part, which is why we are using the job to run it in the back ground. Note that, in this method, we also prepare the `result` already in a String that will be shown to the user. But, there should not be any user dialog in the `run` method, since this method is run in a separate thread in the background – and user dialogs would require to be called from a dedicated GUI thread.

`showResult()` This is the method that will be called for showing the result to the user. And, the infrastructure from `AbstractEPNKJob` will make sure that it will be called from the dedicated GUI thread again. Therefore, we can use all eclipse dialogs for showing the result. Listing 4.14 shows the implementation of this method. The result String, which was prepared during the `run` method is shown to the user by initiating an information dialog.

Listing 4.14: Code for showing the final result

```
protected void showResult() {
  MessageDialog.openInformation(
      null,
      "ePNK: Model checker",
      result
  );
}
```

`canceling()` This method is a call-back mechanism that allows the user to abort jobs. In the case of computation intensive jobs such as model checking, to abort the computation and not to let that thread continue in the background is very important; otherwise this thread would consume all the computation power until it finishes on its own – which could take extremely long. Therefore, MCiE provides a mechanism for aborting model checking operations on some model, by invoking `abort()` – from a different thread of course. Our implementation of the `canceling()` method invokes this `abort()` method to actually finish the model checking – possibly with some delay. This is shown in Listing 4.15, where `transitionsystem` is the one that was constructed in the `prepare` method and on which the model checking is done.

Listing 4.15: Code for aborting the model checking job

```
protected void canceling() {
  if (transitionsystem != null) {
    transitionsystem.abort();
  }
}
```

Together, the `ModelcheckingAction` and the `ModelcheckingJob`, plugged in via the "plugin.xml" implement the complete model checker. It can be invoked in the run-time workbench, by right-clicking on a Petri net of type PTNet, and then selecting "ePNK"→ "Model checker". Then, a user dialog asks for the formulas that should be checked. After that, the model checking runs as a background job: eclipse indicates a running (non-system) job at the bottom of the workspace by a small icon with a running bar in the "progress area". Once the job is finished, the icon will show up with an exclamation mark. When the user clicks on it, the result dialog is shown. If the eclipse progress area and icon are too small for you, you can also open the eclipse progress view ("Window"→ "Show View"→ "Other..." and then select "Progress" in category "General"). This will show all currently running jobs, and the ones that are finished. When clicking on a finished job, the result is shown. Jobs that are still running, can be aborted by pressing the red "abort" button.

For the syntax of the temporal formulas, we refer to the documentation of the MCiE library `http://www2.cs.uni-paderborn.de/cs/kindler/Lehre/MCiE/` and its example formulas or have a look into the documentation of the parser package. Places will be represented as MCiE variables in the formula. But you need to make sure that places in the Petri net have legal MCiE variable names (in particular, there should not be white spaces or special characters in them). One speciality is that the binary temporal operators such as EU, AR, are represented in infix notation like $p1EUp2$ and not in the more usual notation $E[p1Up2]$.

4.3 Adding Petri net types

As mentioned several times already, it is one of the main features of the ePNK that new Petri net types can be plugged in. In this section, we discuss how to plug in a new Petri net type. In Sect. 4.3.1, we start with a simple version, for which we, basically, need to provide a model with the extensions only; as an example, we use P/T-Systems (PTNet), which come as an integral part of the ePNK; but it is defined with ePNK's type definition mechanism.

For more complex Petri net types, we can also define the mapping from the concepts of the Petri net type in the model to the XML representation; and, for Petri nets with structural labels, a *parser* and a *linker* for the labels must be provided. The parser is needed to convert textual label in its concrete syntax to the structure; the linker is needed for linking the use of symbols in some label to their definition in others. We use the example of high-level Petri nets (HLPNG) for discussing the relevant details in Sect. 4.3.2.

4.3.1 Simple Petri net type definitions: PTNet

The definition of P/T-Systems follows almost exactly the general idea outlined already in Sect. 2.2.2 and the ecore model that we use in the Petri net type definition, is almost a copy of the one that we have seen in Fig. 2.2 on page 7 already. It remains to discuss some of the differences in these models, and to discuss the steps to make the type known to the ePNK (in short to plug it into the ePNK).

The model

The Petri net type `PTNet` is defined in the plugin project
`org.pnml.tools.epnk.pntypes`.
The main part is the ecore model in "PTnet.ecore", where the diagram information is contained in "PTNet.ecorediag". You can open this diagram

by double-clicking on it in the resource explorer[8] (see Sect. 4.1.2). The diagram is shown in Fig. 4.2.

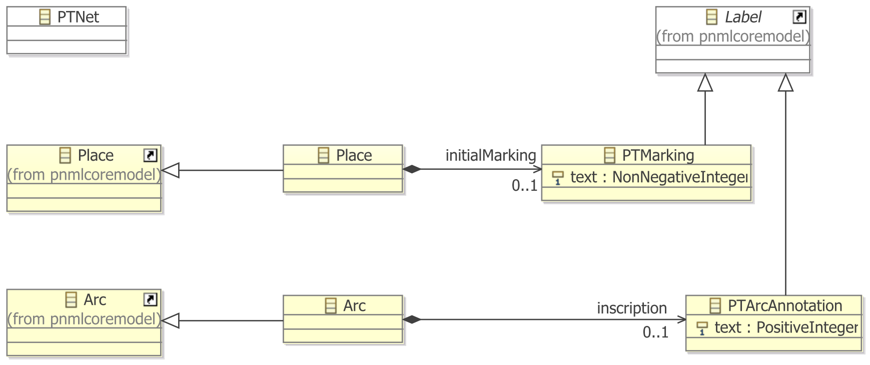


Figure 4.2: The ecore model for PTNet

There are only some minor, but important differences, to the model from Fig. 2.2 on page 7. We discuss these differences below:

1. There is a class `PTNet` in the ecore model, which did not exists in the conceptual model. The reason is that packages are not very tangible in programming and in the eclipse plugin mechanisms. Therefore, we define a Petri net type as an explicit class, which inherits from `PetriNetType` from the PNML core model (`pnmlcoremodel`). It is the class `PTNet`, which will be plugged in as the extension to the ePNK later. Moreover, this class implements some methods that help the ePNK to access the information about its labels; the details, however, do not need to concern us right now.

2. There are two new classes `Place` and `Arc`, which inherit from the classes `Place` and `Arc` from the `pnmlcoremodel`. And it is these new classes to which the additional labels are attached. The reason for this is, that ecore does not have the concept of merging packages. Instead, the extended information for the specific Petri net type is attached to the derived classes in this new package. There could be also a class for `Page` and `Transition`, but we do not need them here, since in P/T-Systems only places and arcs need to have additional labels.

[8]If for some reason, you have opened this diagram with another editor, you can open it with the diagram editor again by selecting the file, right-clicking on it and then selecting "Open With" → "Ecore Diagram Editing".

Note that the name of these two classes are the same as in the PNML core package, which is not ambiguous since the new classes are in a new package. For now, we assume that the names of these classes are the same as in the PNML core model[9].

3. The additional classes for labels, `PTMarking` and `PTAnnotation`, are attached to the new class `Place` and `Arc` as in the conceptual model via a composition – only the directive "refines" is missing, due to the missing concept of merging packages in ecore. The features `text` are directly represented as an attribute of type `NonNegativeInteger` and `PositiveInteger`, which are predefined data types of the ePNK that represent the respective XML Schema data types. The cardinality for the `text` attributes is 1 in both cases – the same as in the conceptual model.

4. The new labels `PTMarking` and `PTAnnotation` are derived from the PNML core model class `Label` and not, as in the conceptual model, from `Annotation`. The reason is that up to now, the ePNK does not distinguish between attribute and annotation labels.

5. A last difference is that there is no OCL constraint in this model. Constraints will be plugged in in a different way in the ePNK as a EMF constraint (see Sect. **??**).

Such a diagram and model can be created by the "Ecore Tools", which will not be discussed here (see the "EMF Ecore Tools Developer Guide" in the "Eclipse Help" and the web pages for some information).

Generating the code

Also the code generation from that ecore model follows the EMF standard procedure. But, we will briefly go through the process of generating all the relevant code below.

Before we can generate the code from the model, we need to create the so-called *generator model* ("genmodel"). This generator model will contain some information on how the code should be generated. For example, the "genmodel" contains the information to which project and which packages, the java code for the model should be generated. The "genmodel" also allows us to configure the generation of the EMF tree editor; for example, we

[9]In principle, the names could be different; but this would require some extra programming, which we do not discuss in this manual for now.

can state whether a features can be changed, whether it should be shown as a child element or as a property, etc. See [2] for more details. A new "genmodel" can be created from an ecore model by selecting the ecore model ("PTnet.ecore" in our case), clicking the right mouse button, and selecting "New"→"Other..." in the pop-up menu and then, in the "New" dialog choosing "EMF Generator Model" in the category "Eclipse Modeling Framework". In the case of a new Petri net type, we will have references to other models like the PNML core model and their "genmodel"; in the wizard for creating the "genmodel", make sure that you do not chose these other models as root models, but that you add (and select) the respective generator models in the lower part as "Referenced Generator models" instead. You do not need to make any changes in the "genmodel", but we recommend that you change the "Base package" property to some reasonable path.

From the "genmodel", we can create the code for the model (*model code*), and the code with the infrastructure for all editors, which is called *edit code*. But, we do not need to generate the editor code. This is done by opening the "genmodel", and then selecting (after clicking the right mouse button) "Generate Model Code" and "Generate Edit Code". After that, you will find[10] the code for the model in the "src" folder of the project with the model and "genmodel". Moreover, the "plugin.xml" will make the model and its code known to eclipse by an extension `org.eclipse.emf.ecore.generated_package`. The edit code will be generated in a project with the same name extended by a suffix ".edit". We do not need to change anything in the edit code[11].

We need to make a minor change in the model code, though, which is discussed in the next section.

Adding the Petri net type to the ePNK

After the above steps, the code for the new model is known to eclipse. But, the ePNK will not know that it is a Petri net type. To this end, we need to define another extension. Before, we can do that, we need to make a minor change in the automatically generated code. We need to make the constructor of the class that represents the new Petri net type public; in our example, this concerns the constructor of the class `PTNetImpl`, which can be found in the package `org.pnml.tols.epnk.pntypes.ptnet.impl`.

Listing 4.16 shows this class. You can see the constructor, which is

[10]If you do not say otherwise in the "genmodel".

[11]In the `org.pnml.tols.epnk.pntypes.edit` project with the "edit code" for PTNets, some of the automatically generated icons in the folder `icons/obj16` have been replaced by some more appropriate images, but this is just a matter of usability.

Listing 4.16: The class `PTNetImpl` with manual changes

```
package org.pnml.tools.epnk.pntypes.ptnet.impl;

import org.eclipse.emf.ecore.EClass;
import org.pnml.tools.epnk.pnmlcoremodel.impl.PetriNetTypeImpl;
import org.pnml.tools.epnk.pntypes.ptnet.PTNet;
import org.pnml.tools.epnk.pntypes.ptnet.PtnetPackage;


// @generated
public class PTNetImpl extends PetriNetTypeImpl implements PTNet {

  /**
   * @generated NOT
   * @author eki
   */
  public PTNetImpl() {
    super();
  }

  /**
   * @generated
   */
  @Override
  protected EClass eStaticClass() {
    return PtnetPackage.Literals.PT_NET;
  }

  // @generated NOT
  // @author eki
  @Override
  public String toString() {
    return "http://www.pnml.org/version-2009/grammar/ptnet";
  }

}
```

public now. The manual change is indicated by the `@generated NOT` tag[12]. The other manual change is the addition of the `toString()` method. This defines the value of the PNML type attribute for nets for that particular Petri net. Here, we use the one from ISO/IEC 15909-2 for P/T-Systems.

With this public constructor, we can plugin the `PTNetImpl` as a new Petri net type to the ePNK now. To this end, we use the extension point `org.pnml.tools.epnk.pntd` in the "plugin.xml". Listing 4.17 shows the relevant part from the "plugin.xml" file in the project

`org.pnml.tools.epnk.pntypes`.

The attribute `point` refers to the ePNK type definition extension point, the

Listing 4.17: The extension `PTNetImpl`

```
<extension
    id="org.pnml.tools.epnk.pntypes.ptnet"
    name="PTNets"
    point="org.pnml.tools.epnk.pntd">
  <type
      class="org.pnml.tools.epnk.pntypes.ptnet.impl.PTNetImpl"
      description="Place/Transition Nets">
  </type>
</extension>
```

`id` is a unique id for the new type, and `name` is some conclusive name (we use the one from ISO/IEC 15909-2). The type refers to the class that implements the new type; in our example, this is our `PTNetImpl` class. In general, this class that is chosen here must extend the class `PetriNetTypeImpl` from the PNML core model code, and have a public constructor (that is why we needed the manual change). The description can contain a longer description of the new net type – for P/T-Systems, we guessed that no further explanation would be needed.

Adding constraints

As mentioned earlier, it is not allowed in P/T-Systems to have arcs that run from places to places or from transitions to transitions. In the conceptual model of PTNets, this was included as an OCL constraint in the UML model already. In the ePNK, this constraint must be added separately, which is

[12]Actually, we could just delete the tag `@generated`, but it is easier to search for and keep track of manual changes, if they are tagged with `@generated NOT`.

done by the standard mechanisms of EMF Validation in the "plugin.xml".

Listing 4.18 shows the part of the "plugin.xml" that defines this constraint. The actual OCL constraint is defined in the bottom in the XML CDATA part. This OCL expression resembles the one from the conceptual UML model, but is slightly different syntactically – which is due to the specific technology. Moreover the "headline" that states the context `Arc` is missing, since in the EMF Validation technology, the context is explicitly set by the `target` element, which you can find immediately above, which is the `Arc` of the ptnet package (the URL is defined in the model). The declaration of the events is necessary here, since we made this constraint a *live constraint*, which means that the editors will make sure not to violate it during editing. To this end, the editor needs to know the change of which features might violate the constraint; in our example, this is setting the source or the target of an arc.

The rest of this constraint definition is a bit more technical. But, we go briefly through it. The extension that we actually define is a *constraint provider*, which consists of the package it refers to and the constraints. In our case, the package is the PNML core model – even though it is for a specific Petri net types. The reason is that the validation always starts from the PNML core model. The constraints are defined in a category; the ePNK defines its own category, which is used here:

`org.pnml.tools.epnk.validation`

Each constraint must have a unique `id`, must state the language it is defined in (OCL in our example), have a `name`, a `severity`, and a `statusCode`. The status code can be freely chosen; the ePNK uses codes starting with a 3 for constraints concerning Petri net types. The mode can be *live* or *batch*; in live mode, the graphical editor will watch them and not allow edit operations that would violate them – this is what we chose in our example. Other constraints, like correctness of structured labels, might be defined to be in batch mode; the graphical editor will allow for syntactically incorrect labels, but this will be reported when validating the net. The last information in the constraint is a `message`, which is shown to the end user when the constraint is violated. The parameter `{0}` refers to the object that violates the constraint (in its String representation) – for constraints other than OCL, there could be more parameters. Moreover, there is a longer `description` of the constraint.

As mentioned above, the constraint can be formulated in different languages. It could, for example, be in Java, which would require to implement a java class. There are some examples of such java constraints in the HLPNG definition. Java is more convenient for implementing more complex

Listing 4.18: Adding a constraint for PTNets

```
<extension point="org.eclipse.emf.validation.constraintProviders">
  <constraintProvider cache="true">
    <package
      namespaceUri="http://org.pnml.tools/epnk/pnmlcoremodel">
    </package>

    <constraints categories="org.pnml.tools.epnk.validation">
      <constraint
          id=
  "org.pnml.tools.epnk.pntypes.ptnet.validation.PT_TP_ArcsOnly"
          lang="OCL"
          mode="Live"
          name="PT or TP Arcs only"
          severity="ERROR"
          statusCode="301">
        <message>
The arc {0} must run from a place to a transition or vice versa.
        </message>
        <description>
Arcs between two places or transitions are forbidden in
P/T-nets (see Clause 5.3.1 of ISO/IEC 15909-2).
        </description>
        <target
            class="Arc:http://org.pnml.tools/epnk/types/ptnet">
          <event name="Set">
            <feature name="source"/>
            <feature name="target"/>
          </event>
        </target>
<![CDATA[
 ( self.source.oclIsKindOf(pnmlcoremodel::PlaceNode) and
   self.target.oclIsKindOf(pnmlcoremodel::TransitionNode) ) or
 ( self.source.oclIsKindOf(pnmlcoremodel::TransitionNode) and
   self.target.oclIsKindOf(pnmlcoremodel::PlaceNode) )
]]>
      </constraint>
    </constraints>
  </constraintProvider>
</extension>
```

constraints.

Note that we did not define any mapping from the concepts defined in the ecore model of P/T-Systems to their representation in XML. The reason is that, the standard mapping is good enough: the name of the composition in which the label is contained is the XML element, and the text feature of the label is mapped to the XML element `<text>` (see Fig. 2.1 on page 9 for an example). A mapping to XML needs to be defined only when the standard mapping is not enough, or when we have structured labels, which will be discussed in the next section.

4.3.2 Petri net type definitions in general: HLPNG

In this section, we discuss some more advanced mechanisms that can be used for defining new Petri net types. These mechanism will be discussed by the help of the Petri net type definition of high-level Petri nets (HLPNGs). Therefore, we start with an overview of the concepts of HLPNGs, from the implementation point of view (for the conceptual part we refer to Sect. 3.5.2 and for a detailed discussion of all models and concepts, we refer to [5]).

Overview of HLPNGs

As discussed in Sect. 3.5.2, HLPNGs have different kinds of complex labels: *declarations* of variables, sorts, and operators; *types* defining the sort of the tokens of a place, *marking* which are multiset terms defining the initial marking, *conditions* as transition guards, and *arc annotations* that define which tokens are consumed, resp. produced when a transition fires. What is more, the labels cannot be considered isolated from each other any more – some labels, like markings, arc annotations, or conditions may use *symbols* that are defined in other labels – in particular, in the declarations.

Figure 4.3 shows the ecore model defining these concepts of HLPNGs, which can be found in the folder `model` in project[13]

`org.pnml.tools.epnk.pntypes.hlpngs.pntd`

This model follows the same principles as the model for PTNets, which was discussed in Sect. 4.3.1. The main differences are, that the defined Petri net type `HLPNG` extends a more advanced class `StructuredPetriNetType`, and

[13]This is the plugin in which HLPNGs are plugged into the ePNK; since HLPNGs are quite complex, and require many models, and also the implementation of a parser, the underlying definitions are defined in different other projects; all of these projects have a name with prefix `org.pnml.tools.epnk.pntypes.hlpngs` – some of them are generated automatically from models or from a grammar.

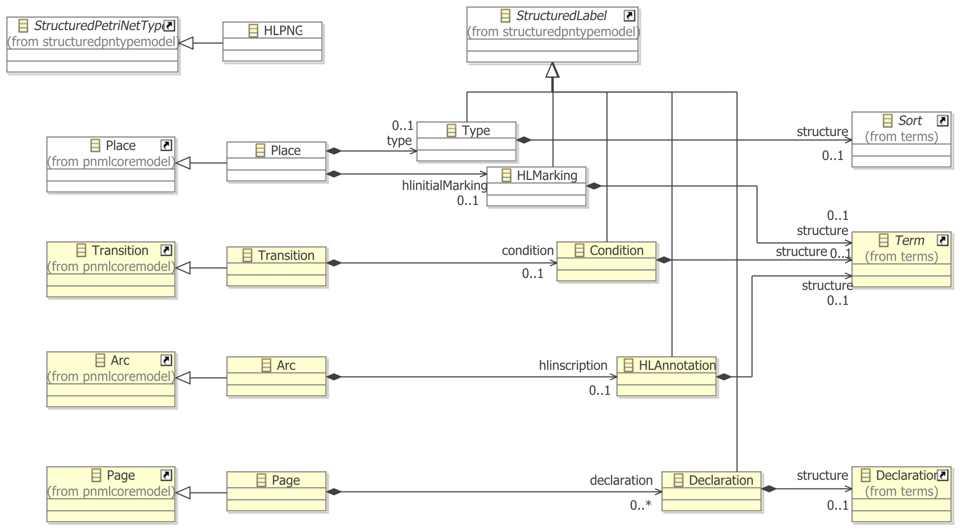
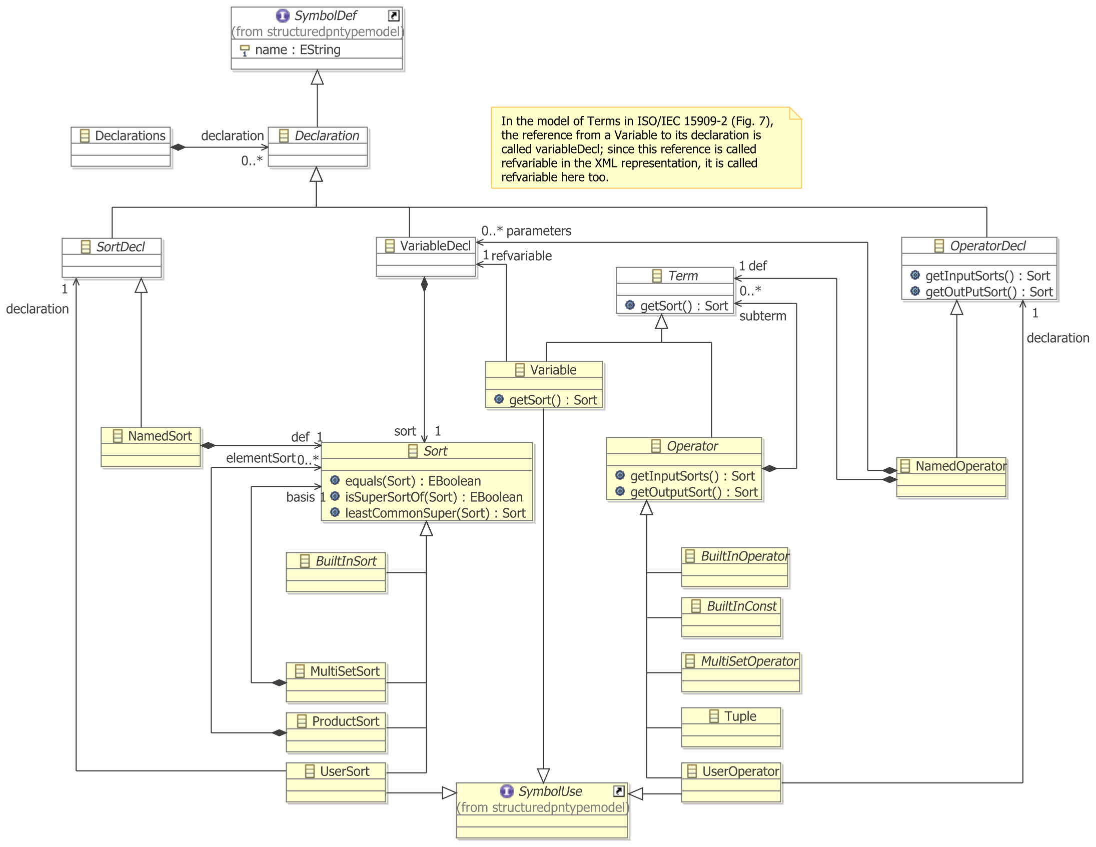


Figure 4.3: The ecore model for HLPNGs

all labels extend `StructuredLabel`, which are part of the PNML core model. These two classes provide the infrastructure needed for parsing the textual labels and for establishing the links between these labels. This structure will be discussed in Sect. 4.3.2.

The actual content of all these labels is defined in their containment `structure`; note that we use `Term` as the contents for `HLMarking`, `Condition`, and `HLAnnotation`, since all of them are terms – just with different additional constraints imposed on them (see Sect. 3.5.2 and Sect. 4.3.2).

The detailed structure and concepts of terms, sorts, and declarations, are defined in several other models. Since these details are not too relevant for understanding the definition of structured Petri net types, we discuss only the main part of that model. This part of the model is shown in Fig. 4.4 – this as well as the diagrams of all the other models can be found in the plugin `org.pnml.tools.epnk.pntypes.hlpngs.datatypes`. For a detailed discussion of these models and their concepts, we refer to [5]. There is only one important difference, which are the classes `SymbolDef` and `SymbolUse`, which do not occur in the models of ISO/IEC 15909-2. These two classes, are the ePNKs infrastructure for dealing with the definition of symbols and their use in a uniform and generic way. They are part of the PNML core model concerning structured Petri net types, which will be discussed in the next section.



In the model of Terms in ISO/IEC 15909-2 (Fig. 7), the reference from a Variable to its declaration is called variableDecl; since this reference is called refvariable in the XML representation, it is called refvariable here too.

Figure 4.4: The ecore model for the main concepts of HLPNGs

Structured Petri net types and structured labels

As mentioned above, the ePNK provides some general interfaces and infrastructure for defining structured Petri net types that extracts the general concepts of more complex types. This is, again captured in models (and the code generated from them).

The model for structured Petri net types can be found in the `model` folder of the ePNK core project `org.pnml.tools.epnk`:

`PNMLStructuredPNTypeModel`

The diagram is shown in Fig. 4.5. We know the classes `PetriNetType` and `Label` as well as the interface `ID`, which is used for all ePNK elements that have an id, already from the PNML core model. The abstract class `StructuredLabel` extends the class `Label`, it has an attribute `text`, which

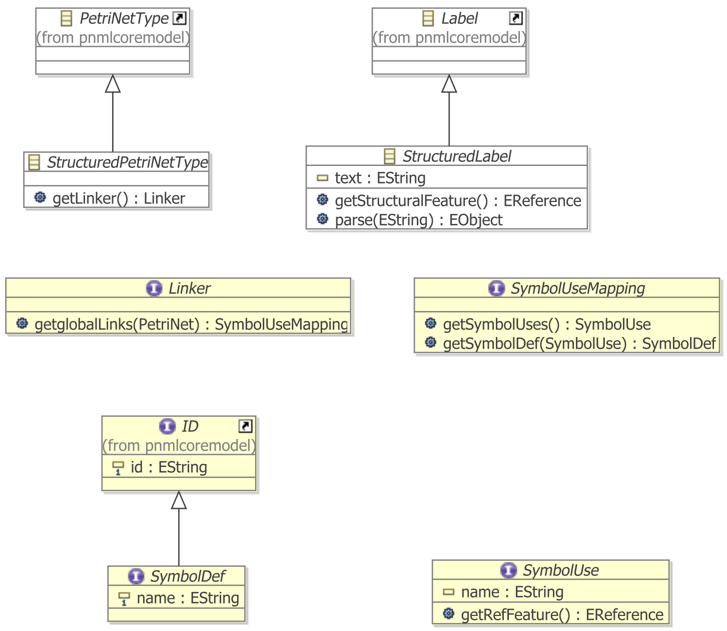


Figure 4.5: The model for structured Petri net types

stores the content of this label as a text String. The actual structural contents will be defined by classes that extend it (we have seen some examples in Fig. 4.3 already). Since, the ePNK cannot not know these concrete implementations, classes extending the structural label must make the reference to this structural contents known to the ePNK. This is achieved by the method `getStructuralFeature()`; as long as the feature for the structure is called 'structure" in the model, we do not need to do anything in the implementation (the ePNK will access this feature in a reflective way); only if for some reason, the model chooses a different name, this method must be implemented manually. Moreover, every class for a structural label must provide a method for parsing a String – a representation of this label in concrete syntax; an implementation of this method may return null, if the text cannot be parsed, or it must return some object (to be precise an `EObject` which is the EMF version of objects) with all the substructure of that label. In particular, that object must have a type that is compatible with the labels structural feature. This method must be implemented manually for every new extension since the ePNK cannot guess the concrete syntax.

The abstract class `StructuredPetriNetType` has one additional method, which must provide a `Linker` for linking the uses of some symbols to their definitions, which are captured by classes `SymbolDef` and `SymbolUse`. A `SymbolDef` is an `ID` and has a name, which will be used to refer to it (the id is internal to PNML and the ePNK). This name will be used in `SymbolUse`, again as attribute name, to refer to the definition. The feature that actually refers to the definition, can be accessed via the method `getRefFeature()`. Since the ePNK, does not know anything about how to make these connections, the Petri net type needs to provide the linker (via the method `getLinker()`). `Linker` is an interface: a single method `getglobalLinks()`, which takes a Petri net and returns a `SymbolUseMapping`, which is also an interface. Conceptually, the `SymbolUseMapping` maps a `SymbolUse` to its definition `SymbolDef`. All the symbol uses for which there exists a mapping, can be obtained (as a list) via the method `getSymbolUse()`; and for each symbol use, the method `getSymbolDef()` will return the definition of that symbol.

With this infrastructure, the ePNK can deal with all kinds of structural labels, provided they implement the required methods. We will have a look at an implementation examples next: We consider the label `Condition` in the Petri net type definition for HLPNGs again (see Fig. 4.3) – the other labels are similar. Its structural feature is the containment `structure` to class `Term`. Since this is the standard name for structured labels, we do not need to override the method `getStructuralFeature`. But, we need to implement the parse method. The parsers for all labels of HLPNGs were automatically generated by Xtext, and made available in a singleton class `HLPNGParser` in package

`org.pnml.tools.epnk.pntypes.hlpngs.datatypes.concretesyntax`

in a project with the same name. For parsing a term, `HLPNGParser` provides a method `parseTerm(String)`. This singleton and its method `parseTerm` is used in the implementation of `ConditionImpl` (you will find it in the package

`org.pnml.tools.epnk.pntypes.hlpng.pntd.hlpngdefinition.impl`

in project `org.pnml.tools.epnk.pntypes.hlpng.pntd`).

Since linking is across all the different labels of a net, there is only a single linker for every net. For HLPNGs, this is implemented in package

`org.pnml.tools.epnk.pntypes.hlpngs.datatypes.concretesyntax.linking`

in project

`org.pnml.tools.epnk.pntypes.hlpngs.datatypes.concretesyntax`

This class is `HLPNGLinker`; basically it goes through the complete Petri net twice; in the first round, it creates a symbol table of all symbol definitions; in the second round, this symbol table will be used to look up the definitions for every symbol use, which is stored in the `SymbolMapping`, which implements the `SymbolUseMapping` that we discussed above.

To make this linker known to the ePNK, the class `HLPNGImpl` in package `org.pnml.tools.epnk.pntypes.hlpng.pntd.hlpngdefinition.impl` implements the method `getLinker()`: it returns an instance of `HLPNGLinker`. Note that, in the class `HLPNGImpl`, we also need to make the constructor public and implement the `toString()` method (as discussed in Sect. 4.3.1).

Constraints

For HLPNGs, we needed to implement quite many constraints. As an example for a java constraint, we discuss just one example here. The rest of them would not provide much insight into the mechanisms of the ePNK – though they might provide some insights to the inner workings of HLPNGs themselves. There is also one OCL constraint, which forbids connecting places with places and transitions with transitions. But, this exactly as for PTNets, which is why we do not discuss it here again.

All constraints for HLPNGs are defined in the project

`org.pnml.tools.epnk.pntypes.hlpng.pntd`,

the java constraint implementations can be found in the package

`org.pnml.tools.epnk.pntypes.hlpng.pntd.validation`

We discuss the constraint that transition conditions must have type boolean, which is implemented in class `ConditionIsBoolType`. Listing 4.19 shows this class. This constraint extends the class `AbstractModelConstraint` from EMF Validation and implements the method `validate`. From the validation context, it obtains the target object, which should be a transition (see later). But, we are defensive and check that explicitly. Then, we obtain the condition label of that transition, and if it is not null, get the term (its structure). Then, we check whether the sort of the term is boolean[14]. If it is, we return a failure status via the validation context, and add the transition and the textual label to an array of objects (which is used in the error message to be defined later). Otherwise, we return a success status. Note that the EMF Validation Framework makes sure that this validate method is called for all transitions of a selected Petri net, Petri net document or page, once it is properly plugged in, which is discuss below.

Plugging in a java constraint is very similar to plugging in OCL constraints. The relevant fragment of the "plugin.xml" is shown in Listing 4.20. The main differences are that the attribute `lang` is "Java" now and the at-

[14]The implementation of `getSort()` for terms is actually quite complex, but we do not discuss that here.

Listing 4.19: The constraint that conditions have type boolean

```
package org.pnml.tools.epnk.pntypes.hlpng.pntd.validation;

import org.eclipse.core.runtime.IStatus;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.validation.AbstractModelConstraint;
import org.eclipse.emf.validation.IValidationContext;
import
  org.pnml.tools.epnk.pntypes.hlpng.pntd.hlpngdefinition.Condition;
import
  org.pnml.tools.epnk.pntypes.hlpng.pntd.hlpngdefinition.Transition;
import
  org.pnml.tools.epnk.pntypes.hlpngs.datatypes.booleans.Bool;
import org.pnml.tools.epnk.pntypes.hlpngs.datatypes.terms.Sort;
import org.pnml.tools.epnk.pntypes.hlpngs.datatypes.terms.Term;

public class ConditionIsBoolType extends AbstractModelConstraint {

  public IStatus validate(IValidationContext ctx) {
    EObject object = ctx.getTarget();

    if (object instanceof Transition) {
      Transition transition = (Transition) object;
      Condition condition = transition.getCondition();
      if (condition != null) {
        Term term = condition.getStructure();
        if (term != null) {
          Sort sort = term.getSort();
          if (sort != null) {
            if (!(sort instanceof Bool)) {
              return ctx.createFailureStatus(
                  new Object[] {transition,
                      condition.getText()});
            }
          }
        }
      }
    }
    return ctx.createSuccessStatus();
  }
}
```

Listing 4.20: Adding the constraint for conditions

```
<extension point="org.eclipse.emf.validation.constraintProviders">
  <constraintProvider cache="true">
    <package
      namespaceUri="http://org.pnml.tools/epnk/pnmlcoremodel">
    </package>

    <constraints categories="org.pnml.tools.epnk.validation">
    ...
      <constraint
        lang="Java"
        class="org.pnml. ... .validation.ConditionIsBoolType"
        severity="ERROR"
        mode="Batch"
        name="Condition is of type boolean"
        id="org.pnml. ... .validation.ConditionIsBoolType"
        statusCode="314">
        <target class=
  "Transition:http://org.pnml.tools/epnk/pntypes/hlpng/pntd/hlpng"/>
        <description>
  The condition must be of type BOOL.
        </description>
        <message>
  The condition {1} of transition {0} is not of type BOOL.
        </message>
      </constraint>
    ...
    </constraints>
  </constraintProvider>
 </extension>
```

tribute `class` refers to the class `ConditionIsBoolType`, which was discussed above. As target class, the transitions of HLPNGs are defined (that is why we could assume that the target object is a transition). Another difference is that this is no live constraint, but a batch constrain. This means, that it might be violated during editing, and a violation will only be detected on explicit validation. Since this is a batch constraint, we do not need to declare any events in the target.

Another difference to the OCL constraint is, that we can refer to several parameters in the message now. What the different parameters are, depends on the return value of the validation method. In our case, this was the transition (or its String representation) and the text of the label.

The ellipses ("...") indicate that the constraint that we have discussed here, is just one of many other constraint, which are not discussed here.

XML Mappings

In the sections above, we have discussed how to define a Petri net type and all its concepts and constraints. For saving it in PNML, it is also necessary to define how these concepts are represented in XML – at least if the "standard mappings" do not work.

In this section, we will discuss how these mappings are defined. Conceptually, these mappings are tables (in ISO/IEC 15909-2, these tables are given in Clause 7.3.1). In the ePNK, these "tables are programmed" as part of the new Petri net type[15].

We explain the concepts of these "programmed tables" by discussing some of the mappings for HLPNGs. The tables for a new Petri net type are programmed, by overwriting the method
`registerExtendedPNMLMetaData(ExtendedPNMLMetaData metadata)`
of PetriNetType; the parameter `metadata` represents the table, to which we can add entries when this method is called.

Let us have a look at some examples. Listing 4.21 shows an excerpt of the `registerExtendedPNMLMetaData()` method of the class `HLPNGImpl`. Each of the `metadata.add` statements defines one table entry, which defines the mapping of one specific feature of the ecore model to an XML element (we will see later how to map an ecore attribute to an XML attribute). The three statements shown in Listing 4.21 define how the structure feature of

[15]It might be, that a future version will provide a means to plugin these tables directly in some form; but since "programming the tables" is not too difficult, this does not have a high priority.

Listing 4.21: Mappings for type, marking, and condition extensions

```
public void registerExtendedPNMLMetaData(
    ExtendedPNMLMetaData metadata) {
    ...

  metadata.add(
      HlpngdefinitionPackage.eINSTANCE.getType_Structure(),
      HlpngdefinitionPackage.eINSTANCE.getType(),
      TermsPackage.eINSTANCE.getSort(),
      "structure",
      null,
      HLPNGFactory.getHLPNGFactory());

  metadata.add(
      HlpngdefinitionPackage.eINSTANCE.getHLMarking_Structure(),
      HlpngdefinitionPackage.eINSTANCE.getHLMarking(),
      TermsPackage.eINSTANCE.getTerm(),
      "structure",
      null,
      HLPNGFactory.getHLPNGFactory());

  metadata.add(
      HlpngdefinitionPackage.eINSTANCE.getCondition_Structure(),
      HlpngdefinitionPackage.eINSTANCE.getCondition(),
      TermsPackage.eINSTANCE.getTerm(),
      "structure",
      null,
      HLPNGFactory.getHLPNGFactory());

    ...
}
```

the labels `Type`, the `HLMarking`, and the `Condition` are mapped to the XML element `<structure>`. We discuss the first one, the `Type`, in more detail:

- The first parameter, denotes the feature that is mapped, which is the composition from the class `Type` to the class `Sort` (see Fig. 4.3). These source and target classes are mentioned explicitly as second and third parameter again. We refer to the feature and the two classes via singleton classes that describes the elements of the packages (HLP-NGdefinition and Terms). These *package classes*, provide access to all the classes and features within a package (see [2] for more details). Note that `HlpngdefinitionPackage.eINSTANCE` refers to the package `hlpngdefinition` and `TermsPackage.eINSTANCE` to the package `terms`.

- As mentioned above, the second parameter denotes the class to which the feature belongs (it could be a sub-class of `Type` in principle); this is often called the *container* class.

- The third parameter denotes the class that the feature refers to (this could also be a sub class of `Sort`); this is often called the *object* class.

- The forth parameter, defines the XML representation, the string that will be used as XML element in the serialisation of this feature (in our example "structure").

- The fifth parameter could refer to a XML attribute, that might be necessary for creating an ecore object from the XML element (we will discuss an example later). In most cases, these XML attributes are not needed, since the XML element (and the context in which it occurs) provide enough information for creating the ecore element from it.

- The last parameter refers to a factory that is capable of creating an ecore instance of the respective class from the XML element and – if provided – the XML attribute.

You can see that these table entries can be used in two directions. In the one direction, it tells how to serialise a Petri net to its XML syntax; in the other direction, it tells how to create the model elements from the XML syntax. In the latter case, the factories play an important role. Listing 4.22 shows the interface that all these factories must implement. The methods `canCreateObject` and `createObject` have the same parameters, which basically reflect the entries of the table that we discussed above. Only

Listing 4.22: Interface `Factory`

```
package org.pnml.tools.epnk.pnmlcoremodel.serialisation;

import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.EStructuralFeature;

public interface IPNMLFactory {

  public boolean canCreateObject(
      EStructuralFeature feature,
      Object container,
      String element,
      String attribute,
      IAttributeProvider provider);

  public EObject createObject(
      EStructuralFeature feature,
      Object container,
      String element,
      String attribute,
      IAttributeProvider provider);

  public Object createAttributeObject(
      Object object,
      String attribute,
      IAttributeProvider provider);

}
```

the third and six one (the factory itself) are missing. And, there is an additional parameter (`provider`), which will provide access to the values of all attributes of the currently read XML element (in case the factory needs to read them for creating an the object). The method `canCreateObject` is used to find out whether the factory is able to create an object from the provided information, the `createObject` method is used to actually create it. The `createAttributeObject` is used to create an object for some XML attribute. The implementation of these factories is straight-forward and a bit boring – we do not discuss the details here. You can have a look into the class `HLPNGFactory` in package
`org.pnml.tools.epnk.pntypes.hlpng.pntd.hlpngserialisation.factory`
in the project `org.pnml.tools.epnk.pntypes.hlpng.pntd` to get some inspiration. What is more, with an extension that came into version 0.9.0 of the ePNK, the facory can be set to `null`. In which case the standard mechanism for creating an object of the target class will be used; therefore, we will need factories in very rare and special cases only. Typically, the factory can be set to `null`[16]

Listing 4.23: Mappings of an attribute

```
metadata.addAttributeMapping(
    BooleansPackage.eINSTANCE.getBooleanConstant_Value(),
    BooleansPackage.eINSTANCE.getBooleanConstant(),
    "value",
    HLPNGFactory.getHLPNGFactory());
```

Listing 4.23 shows an example[17] of how a feature of the model can be mapped to an XML attribute. In this example, the value of the boolean constant is mapped to the XML attribute `value`. This is where the method `createAttributeObject` of the factory comes into play.

The discussion above, gives a general idea of how these tables and mappings work. All this, however, could have been achieved with the existing mechanisms of EMF: Extended Metadata. But, the mappings of PNML have some specialities that could not be expressed in the Extended Metadata concepts. Therefore, we introduced yet another extension in the ePNK.

[16]Note that except for two features, which were sued to test this new mechanism, the mappings for HLPNGs have not been updated yet; therefore, you will find factories all over these mappings. But, this has historic reasons only and will eventually be changed (making the mappings more maintainable).

[17]Actually, this is the only one of HLPNGs.

In the rest of this section, we will discuss some of these special situations.

Let us consider the serialisation of the simple term `x‘f(x,x)`, where x is a variable and f a user defined operator. The PNML representation is shown in Listing 4.24. In addition to being a bit verbose, there is one thing that

Listing 4.24: PNML representation of `x‘f(x,x)`

```
<numberof>
  <subterm>
    <variable refvariable="5"/>
  </subterm>
  <subterm>
    <useroperator declaration="1">
      <subterm>
        <variable refvariable="5"/>
      </subterm>
      <subterm>
        <variable refvariable="5"/>
      </subterm>
    </useroperator>
  </subterm>
</numberof>
```

is special about this mapping. There is a XML element `<subterm>` for the association form the top-level term (number of) to its subterm, which are represented as two other XML elements, `<variable>` and `<useroperator>`. The XML element `<subterm>` defines to which feature of the term the XML element that is contained in it should go. The XML element inside (e. g. `<variable>`) defines the type that object should have.

The problem here, is that there is an intermediate XML element that has no object as counter part in the model – it represents an association. We call them association elements. The mapping for these association elements is shown in Listing 4.25. The first entry is actually as we have seen it before. The only difference is that the factory produces an instance of a new class `TermAssoc`, which has the nature of a term but, actually, represents an association to a term. We will discuss that class in more detail later. The two other mappings, define the mapping of variables and user operators to XML, and these are different, since they do not refer to any feature at all. They just refer to a container class and a contained class. The container class is the class `TermAssoc`, which will make sure that the variable resp. user operator will be added to the subterm feature of the operator on the

Listing 4.25: Mappings of associations to XML elements

```
metadata.add(TermsPackage.eINSTANCE.getOperator_Subterm(),
    TermsPackage.eINSTANCE.getOperator(),
    TermsPackage.eINSTANCE.getTerm(),
    "subterm",
    null,
    HLPNGFactory.getHLPNGFactory());

metadata.add(null,
    HlpngserialisationPackage.eINSTANCE.getTermAssoc(),
    TermsPackage.eINSTANCE.getVariable(),
    "variable",
    null,
    HLPNGFactory.getHLPNGFactory());

metadata.add(null,
    HlpngserialisationPackage.eINSTANCE.getTermAssoc(),
    TermsPackage.eINSTANCE.getUserOperator(),
    "useroperator",
    null,
    HLPNGFactory.getHLPNGFactory());
```

level above[18].

The class `TermAssoc` does not need to be programmed. This class, as well as the other classes for representing associations, could completely be generated from a model. This model is shown in Fig. 4.6. These classes extend a specific class of our model (the one to which the respective association should go), and the general class for `AssocClass`, which is defined by the ePNK, and implements all the necessary functionality. Note that these

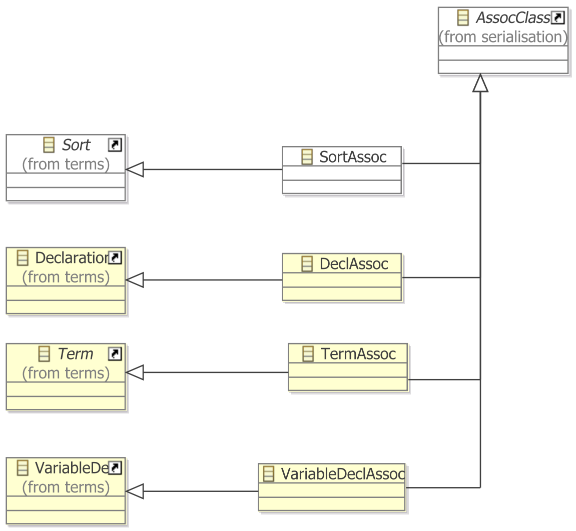


Figure 4.6: The package `hlpngserialisation`

classes will not occur in the model anymore, once it is completely loaded – they are only used when a PNML file is loaded.

In the case of subterms, every subterm occurs in a separate `<subterm>` element – even if a term has several subterms, there is one subterm element for each of them (see Listing **??**). In the case of parameters of an operation declaration, this is different: Listing 4.26 shows the PNML representation of the declaration of a named operator `f(x:INT, y:INT) = x * y`. Here, all variable declarations occur in the same `<parameter>` element. We called these *bundled association elements*. The table entries for this mapping are shown in Listing 4.27. The first one, is almost the same as for association elements, and the Factory `HLPNGFactory` would create an instance of `VariableDeclAssoc` for an XML element `<parameter>`. The new last pa-

[18]There would actually be another way of doing this, in a slightly more elegant way when using "standard features", which will be discussed later in this section.

Listing 4.26: PNML structure for declaration `f(x:INT, y:INT) = x * y`

```
<namedoperator id="1" name="f">
  <parameter>
    <variabledecl id="2" name="x">
      <integer/>
    </variabledecl>
    <variabledecl id="3" name="y">
      <integer/>
    </variabledecl>
  </parameter>
  <def>
    <mult>
      <subterm>
        <variable refvariable="2"/>
      </subterm>
      <subterm>
        <variable refvariable="3"/>
      </subterm>
    </mult>
  </def>
</namedoperator>
```

rameter `true` says, that this is a bundled association. The second table

Listing 4.27: Mapping bundled association elements

```
metadata.add(
  TermsPackage.eINSTANCE.getNamedOperator_Parameters(),
  TermsPackage.eINSTANCE.getNamedOperator(),
  TermsPackage.eINSTANCE.getVariableDecl(),
  "parameter",
  null,
  HLPNGFactory.getHLPNGFactory(),
  true);

metadata.add(
  null,
  null,
  TermsPackage.eINSTANCE.getVariableDecl(),
  "variabledecl",
  null,
  HLPNGFactory.getHLPNGFactory());
```

entry defines the mappings for variable entries, which is independent of the context, which is why the first to parameters are `null`. We call this a *context independent element mapping.*

This context independent element mapping can be applied in any other context. In combination with another special case of mappings which we call *standard feature*, this is a very powerful mechanism. For example, for `Declarations` and sub-element for which context element mapping exists (in the example, there would be variable declarations, sort declarations, and operator declarations), all these elements should be added to this standard feature. The table entry shown in Listing 4.28 defines the composition `declaration` as the standard feature of `Declarations`. Note that there

Listing 4.28: Defining a standard feature

```
metadata.add(TermsPackage.eINSTANCE.getDeclarations_Declaration(),
  TermsPackage.eINSTANCE.getDeclarations(),
  TermsPackage.eINSTANCE.getDeclaration(),
  null,
  null,
  null);
```

is no mapping to XML here. A standard feature of an element just says that, whenever comes from some context independent element, which is not mapped explicitly to a feature, should be added to this standard feature of the model. Of course, there should only be one standard feature – otherwise there would be some ambiguities.

4.4 Adding tool specific information

As discussed in Sect. 2.2.1, the PNML allows tool specific information to be added to all elements of Petri nets – indicated by a special XML element `<toolspecific>`. The ePNK will read and write any tool specific information, and, in principle, its context could be accessed and modified via the class `AnyType`, which is defined in the plugin `org.eclipse.emf.ecore`. But this is tedious and, basically, means navigating in the elements XML structure.

Therefore, the ePNK provides an extension point to plugin tool specific extensions, so that they can accessed and modified via an API specific to the extension which can be defined in terms of a model. We will discuss how to use this extension point by the help of an example: the token positions, which are part of the standard. We have seen an example already in Fig. 2.3 and Listing 2.1 on page 9.

This tool specific extension is defined in the project

`org.pnml.tools.epnk.toolspecific.tokenpositions`

Most of this code and the "plugin.xml" is automatically generated from the model "Tokenpositions.ecore" in the folder "model". This model is shown in Fig. 4.7. The new classes are `PNMLToolInfo` and `Tokengraphics`. The class `PNMLToolInfo` represents the actual tool specific information: it must implement the PNML core model interface `ToolInfo`. The actual contents of this tool specific information is `Tokengraphics`, which consists of one or many coordinates; the class `Coordinate` comes from the PNML core model.

From this model, the code can be generated in the same way as describes in Sect. 4.3.1. First, the "genmodel" must be created, and from the "genmodel", the model code and the edit code must be generated.

After the code generation, the only thing left to plug in the new tool specific extension is to manually create a factory for this tool specific extension, and use this factory for plugging it into the ePNK. The factory for our extension is shown in Listing 4.29. The factory implements the ePNK interface `ToolspecificExtensionFactory`, which consists of four methods. The two methods `createToolInfo` create an instance of this tool specific

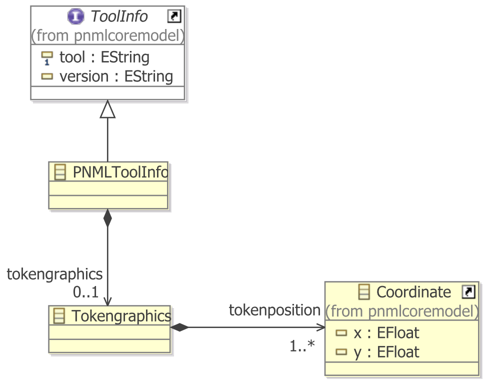


Figure 4.7: The model for tool specific extension tokenpositions

extension; the method with the two String parameters, `tool` and `version` is used, when there is the tool name and a version given, which might return instances of different classes – in our example, however, the version number is irrelevant. The two other methods, must return the tool name for that extension and its version, which, in our example, are encoded in constants.

Listing 4.30 shows the fragment of the "plugin.xml" that is needed to plug in the token position extensions to the ePNK. In addition to the name and the id, the attribute `class` defines to the factory for the tool specific extensions; this class must implement the interface `ToolspecificExtensionFactory`. Moreover, there is a brief description of this extension.

Note that there is no way of explicitly defining the XML syntax of these extensions. The standard XMI serialisation will be used. Eventually, there will be a mapping mechanism similar to the one for Petri net types.

4.5 Overview of the ePNK and its API

In this section, we give a brief overview of the different parts of the ePNK, its project structure and the API. As mentioned earlier, developers would probably not need to change anything in these projects, but the overview helps to better understand the ideas behind the ePNK, the necessary dependencies (that need to be included in new projects via the "plugin.xml") and the functions that are available in the ePNK API.

...

Listing 4.29: Factory for the tool specific extension

```
package org.pnml.tools.epnk.toolspecific.tokenpositions.factory;

import org.pnml.tools.epnk.pnmlcoremodel.ToolInfo;
import org.pnml.tools.epnk.toolspecific.extension.ToolspecificExtensionFactory;
import org.pnml.tools.epnk.toolspecific.tokenpositions.TokenpositionsFactory;


public class TokenpositionsExtensionFactory
    implements ToolspecificExtensionFactory {

  private final static String toolname =     "org.pnml.tool";
  private final static String toolversion =  "1.0";

  public ToolInfo createToolInfo(String tool, String version) {
    // ToolInfo object does not depend on these values:
    return createToolInfo();
  }

  public ToolInfo createToolInfo() {
    return TokenpositionsFactory.eINSTANCE.createPNMLToolInfo();
  }

  public String getToolName() {
    return toolname;
  }

  public String getToolVersion() {
    return toolversion;
  }

}
```

Listing 4.30: Plugging in the token position extension

```
<extension
  id="org.pnml.tools.epnk.toolspecific.tokenpositions"
  name="Token Positions"
  point="org.pnml.tools.epnk.toolspecific">
  <type
    class="org.pnml. ... .factory.TokenpositionsExtensionFactory"
    description="The tool specific extension for token positions">
  </type>
</extension>
```

4.6 Deploying extensions

In this section, we will briefly discuss how own extensions of the ePNK could be deployed, so that others can use it. We will briefly touch on the eclipse *feature* mechanism for combining a bunch of related plugins into a deployable unit, the feature. Moreover, we will briefly discuss how to create an eclipse *update site*, from which the extensions can be installed.

Since this is standard eclipse, however, this will come at a later stage. For now, looking up the keywords "feature" and "update site" in the eclipse help (or googling for them) must be enough.

4.7 Future plans

In this section, we give an overview of some extensions that are planned for the ePNK, which will increase its flexibility, make it easier to use, and provide additional extension mechanisms. The order indicates the priorities, and might be roughly the order in which the features are implemented:

- Full support all graphical features of PNML in the graphical editor (maybe except bezier curves for arcs, which needed to be implemented manually, since GMF does not support them).

- Add a toolspecific feature that exactly saves the graphics of a Petri net as it appears in the graphical ePNK editor (even if no PNML representation exists).

- Smoother integration of EMF-tree editor with the GMF-editor.

- Add the attributes mechanism of PNML together with a mechanism to define the graphical appearance of net objects.
- Implement the serialisation of the structural labels of HLPNGs to concrete syntax.
- Define an extension point for defining the concrete syntax of HLPNGs.
- Portation of the code-generation for extended P/T-Nets to the ePNK (and define the extended Petri net type for that).
- Implement an explicit ePNK extension point for functionality, in order to unify the access to ePNK functions.
- Define an extension point for the XML mapping that directly takes a table instead of “programming the table”.
- Long-term: Support modular PNML.

Chapter 5

Inside the ePNK

In this section, we will have a closer look into the ePNK. We will see how some of the features of the ePNK are implemented and in which way some of the EMF technology needed to be extended to cater for the mappings from the ecore models to XML. Note that this chapter is only for very interested readers, who eventually might contribute to the development of the ePNK or who want to learn a bit more about EMF and how some of EMF's more advanced features work.

For using the ePNK, and for defining some extensions at the ePNK's predefined extension points, you would not need to read this chapter.

...

Chapter 6

PNML: Observations and suggestions

During the work on the ePNK, several issues came up concerning PNML. In this chapter, these issues will summarized and some suggestions on how to improve future versions of PNML will be made.

Bibliography

[1] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, technology, and tools. In W. van der Aalst and E. Best, editors, *Application and Theory of Petri Nets 2003, 24th International Conference*, volume 2679 of *LNCS*, pages 483–505. Springer, June 2003.

[2] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, 2nd edition edition, Apr. 2006.

[3] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

[4] E. Clayberg and D. Rubel. *Eclipse Plug-ins*. The Eclipse Series. Addison-Wesley, 3rd edition edition, 2008.

[5] L. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Treves. A primer on the Petri Net Markup Language and ISO/IEC 15909-2. In K. Jensen, editor, 10th *Workshop on Coloured Petri Nets (CPN 09)*, pages 101–120, Oct. 2009.

[6] L.-M. Hillah, F. Kordon, L. Petrucci, and N. Trèves. PNML Framework: An extendable reference implementation of the petri net markup language. In J. Lilius and W. Penczek, editors, *Petri Nets*, volume 6128 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 2010.

[7] M. Jüngel, E. Kindler, and M. Weber. The Petri Net Markup Language. *Petri Net Newsletter*, 59:24–29, Oct. 2000.

[8] M. Jüngel, E. Kindler, and M. Weber. Towards a generic interchange format for Petri nets – position paper. In R. Bastide, J. Billington,

E. Kindler, F. Kordon, and K. H. Mortensen, editors, *Meeting on XML/SGML based Interchange Formats for Petri Nets*, pages 1–5, June 2000.

[9] E. Kindler. Der Petrinetz-Kern: Ein Traum wird wahr. In H. Ehrig, W. Reisig, and H. Weber, editors, *Move-On-Workshop der DFG-Forschergruppe Petrinetz-Technologie*, pages 121–124. Technische Universiät Berlin, 1997.

[10] E. Kindler. The Petri Net Markup Language and ISO/IEC 15909-2: Concepts, status, and future directions. In E. Schnieder, editor, *Entwurf komplexer Automatisierungssysteme, 9. Fachtagung*, pages 35–55, May 2006. invited paper.

[11] E. Kindler and J. Desel. Der Traum von einem universellen Petrinetz-Werkzeug — Der Petrinetz-Kern. In J. Desel, A. Oberweis, and E. Kindler, editors, *3. Workshop Algorithmen und Werkzeuge für Petrinetze*, number 341 in Forschungsberichte. Institut AIFB, Universität Karlsruhe, Oct. 1996.

[12] E. Kindler and M. Weber. The Petri Net Kernel – an infrastructure for building Petri net tools. *Software Tools for Technology Transfer*, 3(4):486–497, July 2001.

[13] The Eclipse Foundation. The eclipse platform. http://www.eclipse.org.

[14] M. Weber and E. Kindler. The Petri Net Markup Language. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technologies for Modeling Communication Based Systems*, volume 2472 of *LNCS*, pages 124–144. Springer, 2003.